ЯЗЫК C++
ПРОГРАММИРОВАНИЯ

специальное издание

Бьерн Страуструп
создатель C++

БИНОМ

ИЗДАНИЕ 2010

ЯЗЫК ПРОГРАММИРОВАНИЯ C++

специальное издание

The C++ Programming Language

Special Edition

Bjarne Stroustrup

AT&T Labs
Murray Hill, New Jersey

Addison-Wesley
An Imprint of Addison Wesley Longman, Inc.

Бьерн Страуструп

Язык программирования C++

Специальное издание

Перевод с английского
под редакцией
Н.Н. Мартынова

Москва
Издательство **БИНОМ**
2019

УДК 004.43
ББК 32.973.26-018.1
С83

Бьерн Страуструп

Язык программирования C++. Специальное издание. Пер. с англ. — М.: Издательский дом Бином, 2019 г. — 1136 с.: ил.

Книга написана Бьерном Страуструпом – автором языка программирования C++ – и является каноническим изложением возможностей этого языка. Помимо подробного описания собственно языка, на страницах книги вы найдете доказавшие свою эффективность подходы к решению разнообразных задач проектирования и программирования. Многочисленные примеры демонстрируют как хороший стиль программирования на C-совместимом ядре C++, так и современный объектно-ориентированный подход к созданию программных продуктов.

Книга адресована программистам, использующим в своей повседневной работе C++. Она также будет полезна преподавателям, студентам и всем, кто хочет ознакомиться с описанием языка «из первых рук».

Все права защищены. Никакая часть этой книги не может быть воспроизведена в любой форме или любыми средствами, электронными или механическими, включая фотографирование, магнитную запись или иные средства копирования или сохранения информации без письменного разрешения издательства.

Translation copyright © 2010 by Binom Publishers.
C++ Programming Language, The: Special Edition,
First Edition by Bjarne Stroustrup, Copyright © 2000, All Rights Reserved.
Published by arrangement with the original publisher, Addison Wesley Longman, a Pearson Education Company.

© Addison Wesley Longman,
a Pearson Education Company, 2000
© Издание на русском языке.
Издательство Бином, 2012

ISBN 978-5-7989-0425-9 (рус.)
ISBN 0-201-70073-5 (англ.)

Краткое содержание

https://t.me/it_boooks

Предисловие переводчика и редактора . . . 25
Предисловие автора к третьему русскому изданию . . . 26
Предисловие . . . 29
Предисловие ко второму изданию . . . 31
Предисловие к первому изданию . . . 33

Введение . . . 35

1. Обращение к читателю . . . 37
2. Обзор языка C++ . . . 59
3. Обзор стандартной библиотеки . . . 85

Часть I. Основные средства . . . 111

4. Типы и объявления . . . 113
5. Указатели, массивы и структуры . . . 133
6. Выражения и операторы . . . 155
7. Функции . . . 195
8. Пространства имен и исключения . . . 219
9. Исходные файлы и программы . . . 253

Часть II. Механизмы абстракции . . . 281

10. Классы . . . 283
11. Перегрузка операций . . . 327
12. Наследование классов . . . 371
13. Шаблоны . . . 401
14. Обработка исключений . . . 433
15. Иерархии классов . . . 473

Часть III. Стандартная библиотека . . . 515

16. Организация библиотеки и контейнеры . . . 517
17. Стандартные контейнеры . . . 555
18. Алгоритмы и классы функциональных объектов . . . 607
19. Итераторы и аллокаторы . . . 655
20. Строки . . . 689
21. Потоки . . . 717
22. Классы для математических вычислений . . . 775

Часть IV. Проектирование с использованием C++ 809
23. Общий взгляд на разработку программ. Проектирование 811
24. Проектирование и программирование 849
25. Роли классов 895

Приложения и предметный указатель 923
A. Грамматика 925
B. Совместимость 947
C. Технические подробности 961
D. Локализация 1007
E. Исключения и безопасность стандартной библиотеки 1077

Предметный указатель. 1117

Содержание

https://t.me/it_boooks

Предисловие переводчика и редактора 25
Предисловие автора к третьему русскому изданию 26
Предисловие 29
Предисловие ко второму изданию 31
Предисловие к первому изданию 33

Введение 35

Глава 1. Обращение к читателю 37

1.1. Структура книги 37
1.1.1. Примеры и ссылки 39
1.1.2. Упражнения 40
1.1.3. Замечания о конкретных реализациях языка (компиляторах) 40
1.2. Как изучать C++ 40
1.3. Как проектировался C++ 42
1.3.1. Эффективность и структура 43
1.3.2. Философские замечания 44
1.4. Исторические замечания 45
1.5. Применение C++ 47
1.6. Языки С и C++ 49
1.6.1. Информация для С-программистов 50
1.6.2. Информация для C++-программистов 50
1.7. Размышления о программировании на C++ 51
1.8. Советы 53
1.8.1. Литература 54

Глава 2. Обзор языка C++ 59

2.1. Что такое язык C++? 59
2.2. Парадигмы программирования 60
2.3. Процедурное программирование 61
2.3.1. Переменные и простейшая арифметика 62
2.3.2. Операторы ветвления и циклы 63
2.3.3. Указатели и массивы 64
2.4. Модульное программирование 65
2.4.1. Раздельная компиляция 67
2.4.2. Обработка исключений 68

2.5. Абстракция данных 69
2.5.1. Модули, определяющие типы 69
2.5.2. Типы, определяемые пользователем 71
2.5.3. Конкретные типы 73
2.5.4. Абстрактные типы 74
2.5.5. Виртуальные функции 77
2.6. Объектно-ориентированное программирование 77
2.6.1. Проблемы, связанные с конкретными типами 78
2.6.2. Классовые иерархии 79
2.7. Обобщенное программирование 81
2.7.1. Контейнеры 81
2.7.2. Обобщенные алгоритмы 82
2.8. Заключение 84
2.9. Советы 84

Глава 3. Обзор стандартной библиотеки 85
3.1. Введение 85
3.2. Hello, world! (Здравствуй, мир!) 86
3.3. Пространство имен стандартной библиотеки 87
3.4. Вывод 87
3.5. Строки 88
3.5.1. С-строки 90
3.6. Ввод 90
3.7. Контейнеры 92
3.7.1. Контейнер vector 93
3.7.2. Проверка диапазона индексов 94
3.7.3. Контейнер list 95
3.7.4. Контейнер map 96
3.7.5. Контейнеры стандартной библиотеки 97
3.8. Алгоритмы 98
3.8.1. Использование итераторов 99
3.8.2. Типы итераторов 101
3.8.3. Итераторы и ввод/вывод 102
3.8.4. Алгоритм for_each и предикаты 103
3.8.5. Алгоритмы, использующие функции-члены классов 105
3.8.6. Алгоритмы стандартной библиотеки 106
3.9. Математические вычисления 107
3.9.1. Комплексные числа 107
3.9.2. Векторная арифметика 107
3.9.3. Поддержка базовых вычислительных операций 108
3.10. Основные средства стандартной библиотеки 108
3.11. Советы 109

Часть I. Основные средства 111

Глава 4. Типы и объявления 113
4.1. Типы 113
4.1.1. Фундаментальные типы 114
4.2. Логический тип 115

4.3. Символьные типы 116
4.3.1. Символьные литералы 117
4.4. Целые типы 118
4.4.1. Целые литералы 118
4.5. Типы с плавающей запятой 119
4.5.1. Литералы с плавающей запятой 119
4.6. Размеры 119
4.7. Тип void 121
4.8. Перечисления 122
4.9. Объявления 123
4.9.1. Структура объявления. 125
4.9.2. Объявление нескольких имен 126
4.9.3. Имена 126
4.9.4. Область видимости 127
4.9.5. Инициализация. 129
4.9.6. Объекты и леводопустимые выражения (lvalue) 130
4.9.7. Ключевое слово typedef 130
4.10. Советы 131
4.11. Упражнения 132

Глава 5. Указатели, массивы и структуры. 133
5.1. Указатели 133
5.1.1. Нуль 134
5.2. Массивы. 135
5.2.1. Инициализация массивов 135
5.2.2. Строковые литералы 136
5.3. Указатели на массивы 138
5.3.1. Доступ к элементам массивов 139
5.4. Константы. 141
5.4.1. Указатели и константы 143
5.5. Ссылки 144
5.6. Тип void* 148
5.7. Структуры 149
5.7.1. Эквивалентность типов 152
5.8. Советы 152
5.9. Упражнения 152

Глава 6. Выражения и операторы 155
6.1. Калькулятор 155
6.1.1. Синтаксический анализатор. 156
6.1.2. Функция ввода 161
6.1.3. Низкоуровневый ввод. 163
6.1.4. Обработка ошибок 164
6.1.5. Управляющая программа 165
6.1.6. Заголовочные файлы 166
6.1.7. Аргументы командной строки 167
6.1.8. Замечания о стиле 169
6.2. Обзор операций языка C++ 169
6.2.1. Результаты операций 171

6.2.2. Последовательность вычислений 172
6.2.3. Приоритет операций 173
6.2.4. Побитовые логические операции 174
6.2.5. Инкремент и декремент 175
6.2.6. Свободная память 177
6.2.6.1. Массивы 179
6.2.6.2. Исчерпание памяти 180
6.2.7. Явное приведение типов 181
6.2.8. Конструкторы 182
6.3. Обзор операторов языка C++ 183
6.3.1. Объявления как операторы 184
6.3.2. Операторы выбора (условные операторы) 185
6.3.2.1. Объявления в условиях 187
6.3.3. Операторы цикла 188
6.3.3.1. Объявления в операторах цикла for 189
6.3.4. Оператор goto 189
6.4. Комментарии и отступы 190
6.5. Советы 192
6.6. Упражнения 192

Глава 7. Функции 195

7.1. Объявления функций 195
7.1.1. Определения функций 196
7.1.2. Статические переменные 197
7.2. Передача аргументов 197
7.2.1. Массивы в качестве аргументов 199
7.3. Возвращаемое значение 200
7.4. Перегрузка имен функций 201
7.4.1. Перегрузка и возвращаемые типы 204
7.4.2. Перегрузка и области видимости 204
7.4.3. Явное разрешение неоднозначностей 204
7.4.4. Разрешение в случае нескольких аргументов 205
7.5. Аргументы по умолчанию 206
7.6. Неуказанное число аргументов 207
7.7. Указатели на функции 209
7.8. Макросы 213
7.8.1. Условная компиляция 215
7.9. Советы 216
7.10. Упражнения 217

Глава 8. Пространства имен и исключения 219

8.1. Разбиение на модули и интерфейсы 219
8.2. Пространства имен 221
8.2.1. Квалифицированные имена 223
8.2.2. Объявления using 224
8.2.3. Директивы using 226
8.2.4. Множественные интерфейсы 227
8.2.4.1. Альтернативы интерфейсам 229
8.2.5. Как избежать конфликта имен 231
8.2.5.1. Неименованные пространства имен 232

8.2.6. Поиск имен. 233
8.2.7. Псевдонимы пространств имен 234
8.2.8. Композиция пространств имен 235
8.2.8.1. Отбор отдельных элементов из пространства имен 236
8.2.8.2. Композиция пространств имен и отбор элементов 237
8.2.9. Пространства имен и старый код 238
8.2.9.1. Пространства имен и язык С 238
8.2.9.2. Пространства имен и перегрузка 239
8.2.9.3. Пространства имен открыты 240
8.3. Исключения 241
8.3.1. Ключевые слова throw и catch 242
8.3.2. Обработка нескольких исключений 244
8.3.3. Исключения в программе калькулятора 246
8.3.3.1. Альтернативные стратегии обработки ошибок. 249
8.4. Советы 251
8.5. Упражнения 252

Глава 9. Исходные файлы и программы 253

9.1. Раздельная компиляция 253
9.2. Компоновка (linkage). 254
9.2.1. Заголовочные файлы 257
9.2.2. Заголовочные файлы стандартной библиотеки. 259
9.2.3. Правило одного определения 260
9.2.4. Компоновка с кодом, написанном не на языке C++. 262
9.2.5. Компоновка и указатели на функции 264
9.3. Применяем заголовочные файлы 265
9.3.1. Единственный заголовочный файл 265
9.3.2. Несколько заголовочных файлов 268
9.3.2.1. Остальные модули калькулятора 271
9.3.2.2. Использование заголовочных файлов 273
9.3.3. Защита от повторных включений 274
9.4. Программы 275
9.4.1. Инициализация нелокальных переменных. 275
9.4.1.1. Завершение выполнения программы 276
9.5. Советы 278
9.6. Упражнения 278

Часть II. Механизмы абстракции 281

Глава 10. Классы 283

10.1. Введение 283
10.2. Классы 284
10.2.1. Функции-члены 284
10.2.2. Управление режимом доступа 285
10.2.3. Конструкторы 287
10.2.4. Статические члены 288
10.2.5. Копирование объектов класса 290
10.2.6. Константные функции-члены 290
10.2.7. Ссылки на себя 291
10.2.7.1. Физическое и логическое постоянство. 292
10.2.7.2. Ключевое слово mutable. 294

10.2.8. Структуры и классы . . . 295
10.2.9. Определение функций в теле определения класса. . . 296
10.3. Эффективные пользовательские типы . . . 297
10.3.1. Функции-члены . . . 300
10.3.2. Функции поддержки (helper functions) . . . 302
10.3.3. Перегруженные операции . . . 303
10.3.4. Роль конкретных классов . . . 303
10.4. Объекты . . . 304
10.4.1. Деструкторы . . . 305
10.4.2. Конструкторы по умолчанию. . . . 306
10.4.3. Конструирование и уничтожение объектов . . . 307
10.4.4. Локальные объекты . . . 307
10.4.4.1. Копирование объектов . . . 308
10.4.5. Динамическое создание объектов в свободной памяти . . . 309
10.4.6. Классовые объекты как члены классов . . . 310
10.4.6.1. Обязательная инициализация членов . . . 311
10.4.6.2. Члены-константы . . . 312
10.4.6.3. Копирование членов . . . 313
10.4.7. Массивы. . . . 314
10.4.8. Локальные статические объекты . . . 315
10.4.9. Нелокальные объекты . . . 316
10.4.10. Временные объекты. . . . 318
10.4.11. Размещение объектов в заданных блоках памяти . . . 319
10.4.12. Объединения . . . 321
10.5. Советы . . . 322
10.6. Упражнения . . . 323

Глава 11. Перегрузка операций . . . 327

11.1. Введение . . . 327
11.2. Функции-операции . . . 329
11.2.1. Бинарные и унарные операции. . . . 330
11.2.2. Предопределенный смысл операций . . . 331
11.2.3. Операции и пользовательские типы . . . 331
11.2.4. Операции и пространства имен . . . 332
11.3. Тип комплексных чисел. . . . 334
11.3.1. Перегрузка операций функциями-членами и глобальными функциями 334
11.3.2. Смешанная арифметика. . . . 336
11.3.3. Инициализация . . . 337
11.3.4. Копирование . . . 338
11.3.5. Конструкторы и преобразования типов. . . . 339
11.3.6. Литералы . . . 340
11.3.7. Дополнительные функции-члены . . . 341
11.3.8. Функции поддержки (helper functions) . . . 341
11.4. Операции приведения типов . . . 342
11.4.1. Неоднозначности . . . 344
11.5. Друзья класса . . . 346
11.5.1. Поиск друзей . . . 348
11.5.2. Функции-члены или друзья? . . . 349
11.6. Объекты больших размеров . . . 350

11.7. Важные операции . . . 352
11.7.1. Конструктор с модификатором explicit . . . 353
11.8. Индексирование . . . 355
11.9. Функциональный вызов . . . 356
11.10. Разыменование . . . 358
11.11. Инкремент и декремент . . . 360
11.12. Класс строк . . . 362
11.13. Советы . . . 367
11.14. Упражнения . . . 368

Глава 12. Наследование классов . . . 371

12.1. Введение . . . 371
12.2. Производные классы . . . 372
12.2.1. Функции-члены . . . 375
12.2.2. Конструкторы и деструкторы . . . 376
12.2.3. Копирование . . . 378
12.2.4. Иерархии классов . . . 378
12.2.5. Поля типа . . . 379
12.2.6. Виртуальные функции . . . 381
12.3. Абстрактные классы . . . 384
12.4. Проектирование иерархий классов . . . 386
12.4.1. Традиционные иерархии классов . . . 387
12.4.1.1. Критика . . . 389
12.4.2. Абстрактные классы . . . 390
12.4.3. Альтернативные реализации . . . 393
12.4.3.1. Критика . . . 395
12.4.4. Локализация создания объектов . . . 395
12.5. Классовые иерархии и абстрактные классы . . . 397
12.6. Советы . . . 397
12.7. Упражнения . . . 398

Глава 13. Шаблоны . . . 401

13.1. Введение . . . 401
13.2. Простой шаблон строк . . . 402
13.2.1. Определение шаблона . . . 404
13.2.2. Конкретизация шаблона (template instantiation) . . . 406
13.2.3. Параметры шаблонов . . . 406
13.2.4. Эквивалентность типов . . . 407
13.2.5. Проверка типов . . . 408
13.3. Шаблоны функций . . . 409
13.3.1. Аргументы функциональных шаблонов . . . 410
13.3.2. Перегрузка функциональных шаблонов . . . 411
13.4. Применение аргументов шаблона для формирования различных вариантов поведения кода . . . 414
13.4.1. Параметры шаблонов по умолчанию . . . 415
13.5. Специализация . . . 417
13.5.1. Порядок специализаций . . . 420
13.5.2. Специализация шаблонов функций . . . 420
13.6. Наследование и шаблоны . . . 422

13.6.1. Параметризация и наследование 424
13.6.2. Шаблонные члены шаблонов 424
13.6.3. Отношения наследования 425
13.6.3.1. Преобразования шаблонов 426
13.7. Организация исходного кода 427
13.8. Советы 430
13.9. Упражнения 431

Глава 14. Обработка исключений 433

14.1. Обработка ошибок 433
14.1.1. Альтернативный взгляд на исключения 436
14.2. Группировка исключений 437
14.2.1. Производные исключения 438
14.2.2. Композитные (комбинированные) исключения 440
14.3. Перехват исключений 441
14.3.1. Повторная генерация исключений 441
14.3.2. Перехват любых исключений 442
14.3.2.1. Порядок записи обработчиков 443
14.4. Управление ресурсами 444
14.4.1. Использование конструкторов и деструкторов 446
14.4.2. Auto_ptr 447
14.4.3. Предостережение 449
14.4.4. Исключения и операция new 449
14.4.5. Исчерпание ресурсов 450
14.4.6. Исключения в конструкторах 452
14.4.6.1. Исключения и инициализация членов классов 454
14.4.6.2. Исключения и копирование 454
14.4.7. Исключения в деструкторах 455
14.5. Исключения, не являющиеся ошибками 455
14.6. Спецификация исключений 457
14.6.1. Проверка спецификации исключений 458
14.6.2. Неожиданные исключения 459
14.6.3. Отображение исключений 460
14.6.3.1. Отображение исключений пользователем 460
14.6.3.2. Восстановление типа исключения 461
14.7. Неперехваченные исключения 462
14.8. Исключения и эффективность 464
14.9. Альтернативы обработке ошибок 465
14.10. Стандартные исключения 467
14.11. Советы 469
14.12. Упражнения 470

Глава 15. Иерархии классов 473

15.1. Введение и обзор 473
15.2. Множественное наследование 474
15.2.1. Разрешение неоднозначности 475
15.2.2. Наследование и using-объявление 477
15.2.3. Повторяющиеся базовые классы 478
15.2.3.1. Замещение 479
15.2.4. Виртуальные базовые классы 480

15.2.4.1. Программирование виртуальных базовых классов 482
15.2.5. Применение множественного наследования 484
15.2.5.1. Замещение функций виртуальных базовых классов. 486
15.3. Контроль доступа 487
15.3.1. Защищенные члены классов 489
15.3.1.1. Применение защищенных членов класса 490
15.3.2. Доступ к базовым классам 491
15.3.2.1. Множественное наследование и контроль доступа 492
15.3.2.2. Множественное наследование и контроль доступа 493
15.4. Механизм RTTI (Run-Time Type Information) 493
15.4.1. Операция dynamic_cast. 495
15.4.1.1. Применение dynamic_cast к ссылкам 497
15.4.2. Навигация по иерархиям классов. 498
15.4.2.1. Операции static_cast и dynamic_cast 499
15.4.3. Конструирование и уничтожение классовых объектов. 501
15.4.4. Операция typeid и расширенная информация о типе 501
15.4.4.1. Расширенная информация о типе 502
15.4.5. Корректное и некорректное применение RTTI 504
15.5. Указатели на члены классов. 505
15.5.1. Базовые и производные классы 508
15.6. Свободная память. 509
15.6.1. Выделение памяти под массивы 511
15.6.2. «Виртуальные конструкторы» 511
15.7. Советы 513
15.8. Упражнения 514

Часть III. Стандартная библиотека 515

Глава 16. Организация библиотеки и контейнеры 517

16.1. Проектные решения стандартной библиотеки. 517
16.1.1. Проектные ограничения 518
16.1.2. Организация стандартной библиотеки 520
16.1.3. Непосредственная поддержка языка C++. 523
16.2. Дизайн контейнеров 524
16.2.1. Специализированные контейнеры и итераторы 524
16.2.2. Контейнеры с общим базовым классом. 527
16.2.3. Контейнеры STL. 531
16.3. Контейнер типа vector 533
16.3.1. Типы 533
16.3.2. Итераторы 535
16.3.3. Доступ к элементам 536
16.3.4. Конструкторы 538
16.3.5. Стековые операции 541
16.3.6. Операции над векторами, характерные для списков. 543
16.3.7. Адресация элементов 546
16.3.8. Размер и емкость 547
16.3.9. Другие функции-члены 549
16.3.10. Вспомогательные функции (helper functions). 550
16.3.11. Специализация vector<bool> 550
16.4. Советы 551
16.5. Упражнения 552

Глава 17. Стандартные контейнеры **555**
17.1. Стандартные контейнеры 555
17.1.1. Обзор контейнерных операций 556
17.1.2. Краткий обзор контейнеров 559
17.1.3. Внутреннее представление 560
17.1.4. Требования к элементам контейнеров 561
17.1.4.1. Операция сравнения "<" 562
17.1.4.2. Другие операции сравнения 564
17.2. Последовательные контейнеры 565
17.2.1. Контейнер vector 565
17.2.2. Контейнер list 565
17.2.2.1. Операции splice(), sort() и merge() 566
17.2.2.2. «Головные» операции 568
17.2.2.3. Другие операции 568
17.2.3. Контейнер deque 570
17.3. Адаптеры последовательных контейнеров 570
17.3.1. Стек 571
17.3.2. Очередь 572
17.3.3. Очередь с приоритетом 574
17.4. Ассоциативные контейнеры 576
17.4.1. Ассоциативный массив map 576
17.4.1.1. Типы 577
17.4.1.2. Итераторы 577
17.4.1.3. Индексация 579
17.4.1.4. Конструкторы 581
17.4.1.5. Сравнения 581
17.4.1.6. Специфические для контейнера map операции 582
17.4.1.7. Операции, характерные для списков 584
17.4.1.8. Другие функции 586
17.4.2. Ассоциативный контейнер multimap 587
17.4.3. Ассоциативный контейнер set 588
17.4.4. Ассоциативный контейнер multiset 588
17.5. «Почти контейнеры» 589
17.5.1. Строки типа string 589
17.5.2. Массивы valarray 589
17.5.3. Битовые поля bitset 589
17.5.3.1. Конструкторы 590
17.5.3.2. Побитовые операции 591
17.5.3.3. Прочие операции 592
17.5.4. Встроенные массивы 593
17.6. Создание нового контейнера 594
17.6.1. Контейнер hash_map 594
17.6.2. Представление и конструирование 596
17.6.2.1. Поиск 598
17.6.2.2. Операции erase() и resize() 599
17.6.2.3. Хэширование 600
17.6.3. Другие хэшированные ассоциативные контейнеры 601
17.7. Советы 602
17.8. Упражнения 603

Глава 18. Алгоритмы и классы функциональных объектов 607
18.1. Введение 607
18.2. Обзор алгоритмов стандартной библиотеки 608
18.3. Диапазоны (интервалы) и контейнеры. 613
18.3.1. Входные диапазоны 614
18.4. Классы функциональных объектов 615
18.4.1. Базовые классы функциональных объектов. 616
18.4.2. Предикаты. 617
18.4.2.1. Обзор предикатов 618
18.4.3. «Арифметические» функциональные объекты. 619
18.4.4. Адаптеры («связывающие», для адаптирования функций-членов и указателей на функции, «отрицающие») 620
18.4.4.1. «Связывающие» адаптеры 621
18.4.4.2. Адаптирование функций-членов. 623
18.4.4.3. Версии адаптеров для указателей на функции 624
18.4.4.4. «Отрицатели» 625
18.5. Немодифицирующие алгоритмы. 626
18.5.1. Алгоритм for_each() 626
18.5.2. Алгоритмы поиска 627
18.5.3. Алгоритмы count() и count_if() 629
18.5.4. Алгоритмы equal() и mismatch() 630
18.5.5. Поисковые алгоритмы 631
18.6. Модифицирующие алгоритмы. 632
18.6.1. Копирующие алгоритмы 632
18.6.2. Алгоритм transform() 634
18.6.3. Алгоритм unique() 636
18.6.3.1. Критерии сортировки 637
18.6.4. Алгоритмы замены элементов 638
18.6.5. Алгоритмы удаления элементов 640
18.6.6. Алгоритмы fill() и generate() 640
18.6.7. Алгоритмы reverse() и rotate(). 641
18.6.8. Обмен элементов последовательностей местами 642
18.7. Сортировка последовательностей 643
18.7.1. Сортировка 643
18.7.2. Бинарный поиск. 644
18.7.3. Слияние (алгоритмы семейства merge) 645
18.7.4. Разбиение элементов на две группы (алгоритмы семейства partition) . 646
18.7.5. Операции над множествами 646
18.8. «Кучи» 648
18.9. Нахождение минимума и максимума 648
18.10. Перестановки (permutations) 650
18.11. Алгоритмы в С-стиле 650
18.12. Советы 651
18.13. Упражнения 652

Глава 19. Итераторы и аллокаторы 655
19.1. Введение 655
19.2. Итераторы и последовательности 656
19.2.1. Операции над итераторами. 656

19.2.2. Шаблон iterator_traits 658
19.2.3. Категории итераторов 660
19.2.4. Итераторы для вставок 662
19.2.5. Обратные итераторы 663
19.2.6. Потоковые итераторы 664
19.2.6.1. Буфера потоков 666
19.3. Итераторы с проверкой 668
19.3.1. Исключения, контейнеры и алгоритмы 673
19.4. Аллокаторы (распределители памяти) 674
19.4.1. Стандартный аллокатор 674
19.4.2. Пользовательский аллокатор 678
19.4.3. Обобщенные аллокаторы 680
19.4.4. Неинициализированная память 682
19.4.5. Динамическая память 684
19.4.6. Выделение памяти в стиле языка C 685
19.5. Советы 686
19.6. Упражнения 687

Глава 20. Строки 689

20.1. Введение 689
20.2. Символы 690
20.2.1. Шаблон char_traits 690
20.3. Стандартный строковый шаблон basic_string 692
20.3.1. Типы 693
20.3.2. Итераторы 694
20.3.3. Доступ к элементам (символам) 695
20.3.4. Конструкторы 695
20.3.5. Ошибки 697
20.3.6. Присваивание 698
20.3.7. Преобразование в C-строку 699
20.3.8. Сравнения 701
20.3.9. Вставка 703
20.3.10. Конкатенация 704
20.3.11. Поиск 705
20.3.12. Замена 706
20.3.13. Подстроки 707
20.3.14. Размер и емкость 708
20.3.15. Операции ввода/вывода 709
20.3.16. Обмен строк 709
20.4. Стандартная библиотека языка C 710
20.4.1. C-строки 710
20.4.2. Классификация символов 712
20.5. Советы 713
20.6. Упражнения 714

Глава 21. Потоки 717

21.1. Введение 717
21.2. Вывод 719
21.2.1. Потоки вывода 720

21.2.2. Вывод встроенных типов 722
21.2.3. Вывод пользовательских типов 724
21.2.3.1. Виртуальные функции вывода 725
21.3. Ввод 726
21.3.1. Потоки ввода 726
21.3.2. Ввод встроенных типов 727
21.3.3. Состояние потока 729
21.3.4. Ввод символов 731
21.3.5. Ввод пользовательских типов. 734
21.3.6. Исключения 735
21.3.7. Связывание потоков 737
21.3.8. Часовые (sentries) 738
21.4. Форматирование 739
21.4.1. Состояние форматирования 739
21.4.1.1. Копирование состояния форматирования 741
21.4.2. Вывод целых. 741
21.4.3. Вывод значений с плавающей запятой 742
21.4.4. Поля вывода. 743
21.4.5. Выравнивание полей. 744
21.4.6. Манипуляторы. 745
21.4.6.1. Манипуляторы, принимающие аргументы 746
21.4.6.2. Стандартные манипуляторы ввода/вывода 747
21.4.6.3. Манипуляторы, определяемые пользователем 749
21.5. Файловые и строковые потоки 751
21.5.1. Файловые потоки 752
21.5.2. Закрытие потоков 753
21.5.3. Строковые потоки 755
21.6. Буферирование 756
21.6.1. Потоки вывода и буферы 757
21.6.2. Потоки ввода и буферы 758
21.6.3. Потоки и буферы 759
21.6.4. Буфера потоков 760
21.7. Локализация (интернационализация) 764
21.7.1. Функции обратного вызова для потоков 766
21.8. Ввод/вывод языка С 766
21.9. Советы 770
21.10. Упражнения 771

Глава 22. Классы для математических вычислений 775
22.1. Введение 775
22.2. Числовые пределы 776
22.2.1. Макросы для предельных значений 778
22.3. Стандартные математические функции 778
22.4. Векторная арифметика 780
22.4.1. Конструкторы класса valarray. 780
22.4.2. Индексирование и присваивание в классе valarray 782
22.4.3. Функции-члены 783
22.4.4. Внешние операции и функции 786
22.4.5. Срезы 786

22.4.6. Массив slice_array . . . 789
22.4.7. Временные объекты, копирование, циклы . . . 793
22.4.8. Обобщенные срезы . . . 796
22.4.9. Маски (тип mask_array) . . . 797
22.4.10. Тип indirect_array . . . 798
22.5. Комплексная арифметика . . . 798
22.6. Обобщенные численные алгоритмы . . . 800
22.6.1. Алгоритм accumulate() . . . 801
22.6.2. Алгоритм inner_product() . . . 802
22.6.3. Приращения (incremental changes) . . . 803
22.7. Случайные числа . . . 804
22.8. Советы . . . 806
22.9. Упражнения . . . 807

Часть IV. Проектирование с использованием C++ . . . 809

Глава 23. Общий взгляд на разработку программ. Проектирование . . . 811
23.1. Обзор . . . 811
23.2. Введение . . . 812
23.3. Цели и средства . . . 815
23.4. Процесс разработки . . . 818
23.4.1. Цикл разработки . . . 820
23.4.2. Цели проектирования . . . 822
23.4.3. Этапы проектирования . . . 824
23.4.3.1. Этап 1: отражение концепций классами . . . 825
23.4.3.2. Этап 2: определение операций . . . 828
23.4.3.3. Этап 3: выявление зависимостей . . . 830
23.4.3.4. Этап 4: определение интерфейсов . . . 830
23.4.3.5. Реорганизация иерархии классов . . . 831
23.4.3.6. Использование модельных образцов . . . 832
23.4.4. Экспериментирование и анализ . . . 834
23.4.5. Тестирование . . . 836
23.4.6. Сопровождение и поддержка программ . . . 837
23.4.7. Эффективность . . . 837
23.5. Отдельные аспекты управления проектами . . . 838
23.5.1. Повторное использование кода . . . 838
23.5.2. Масштаб . . . 840
23.5.3. Личности . . . 841
23.5.4. Гибридное проектирование . . . 843
23.6. Аннотированная библиография . . . 845
23.7. Советы . . . 847

Глава 24. Проектирование и программирование . . . 849
24.1. Обзор . . . 849
24.2. Проектирование и язык программирования . . . 850
24.2.1. Отказ от классов . . . 852
24.2.2. Отказ от производных классов и виртуальных функций . . . 854
24.2.3. Игнорирование возможностей статической проверки типов . . . 854
24.2.4. Отказ от традиционного программирования . . . 857
24.2.5. Применение исключительно классовых иерархий наследования . . . 859

24.3. Классы . . . 860
24.3.1. Что представляют собой классы . . . 861
24.3.2. Иерархии классов . . . 862
24.3.2.1. Зависимости внутри иерархии классов . . . 865
24.3.3. Агрегация (отношение включения) . . . 867
24.3.4. Агрегация и наследование . . . 869
24.3.4.1. Альтернатива «включение/наследование» . . . 871
24.3.4.2. Альтернатива «агрегация/наследование» . . . 873
24.3.5. Отношение использования . . . 874
24.3.6. Программируемые отношения . . . 875
24.3.7. Отношения внутри класса . . . 877
24.3.7.1. Инварианты . . . 877
24.3.7.2. Утверждения . . . 879
24.3.7.3. Предусловия и постусловия . . . 882
24.3.7.4. Инкапсуляция . . . 883
24.4. Компоненты . . . 884
24.4.1. Шаблоны . . . 886
24.4.2. Интерфейсы и реализации . . . 888
24.4.3. «Жирные» интерфейсы . . . 890
24.5. Советы . . . 893

Глава 25. Роли классов . . . 895
25.1. Разновидности классов . . . 895
25.2. Конкретные классы (типы) . . . 896
25.2.1. Многократное использование конкретных типов . . . 899
25.3. Абстрактные типы . . . 900
25.4. Узловые классы . . . 903
25.4.1. Изменение интерфейсов . . . 905
25.5. Операции . . . 908
25.6. Интерфейсные классы . . . 909
25.6.1. Подгонка интерфейсов . . . 912
25.7. Дескрипторные классы (handles) . . . 914
25.7.1. Операции в дескрипторных классах . . . 917
25.8. Прикладные среды разработки (application frameworks) . . . 918
25.9. Советы . . . 920
25.10. Упражнения . . . 921

Приложения и предметный указатель . . . 923

Приложение А. Грамматика . . . 925
А.1. Введение . . . 925
А.2. Ключевые слова . . . 926
А.3. Лексические соглашения . . . 927
А.4. Программы . . . 930
А.5. Выражения . . . 930
А.6. Операторы . . . 934
А.7. Объявления . . . 935
А.7.1. Деклараторы . . . 938
А.8. Классы . . . 940
А.8.1. Производные классы . . . 941
А.8.2. Особые функции-члены . . . 941

A.8.3. Перегрузка 942
A.9. Шаблоны 942
A.10. Обработка исключений 943
A.11. Директивы препроцессора 944

Приложение B. Совместимость 947

B.1. Введение 947
B.2. Совместимость C и C++ 948
B.2.1. «Тихие» отличия 948
B.2.2. Код на C, не являющийся C++-кодом 948
B.2.3. Нежелательные особенности 951
B.2.4. Код на C++, не являющийся кодом на C 951
B.3. Старые реализации C++ 953
B.3.1. Заголовочные файлы 954
B.3.2. Стандартная библиотека 955
B.3.3. Пространства имен 955
B.3.4. Ошибки выделения памяти 956
B.3.5. Шаблоны 956
B.3.6. Инициализаторы в операторах for 958
B.4. Советы 958
B.5. Упражнения 960

Приложение C. Технические подробности 961

C.1. Введение и обзор 961
C.2. Стандарт 961
C.3. Символьные наборы 963
C.3.1. Ограниченные наборы символов 963
C.3.2. Escape-символы 964
C.3.3. Расширенные символьные наборы 965
C.3.4. Знаковые и беззнаковые символы 966
C.4. Типы целых литералов 967
C.5. Константные выражения 967
C.6. Неявное преобразование типов 967
C.6.1. Продвижения (promotions) 968
C.6.2. Преобразования 968
C.6.2.1. Интегральные преобразования 968
C.6.2.2. Преобразования чисел с плавающей запятой 969
C.6.2.3. Преобразования указателей и ссылок 969
C.6.2.4. Преобразования указателей на члены классов 970
C.6.2.5. Преобразования в логический тип 970
C.6.2.6. Преобразования «значение интегрального типа — значение с плавающей запятой» 970
C.6.3. Обычные арифметические преобразования 971
C.7. Многомерные массивы 971
C.7.1. Векторы 971
C.7.2. Массивы 973
C.7.3. Передача многомерных массивов в функции 974
C.8. Экономия памяти 975
C.8.1. Битовые поля 976
C.8.2. Объединения 977
C.8.3. Объединения и классы 979

C.9. Управление памятью 979
C.9.1. Автоматическая сборка мусора 980
C.9.1.1. Замаскированные указатели 980
C.9.1.2. Операция delete 981
C.9.1.3. Деструкторы 982
C.9.1.4. Фрагментация памяти 982
C.10. Пространства имен 983
C.10.1. Удобство против безопасности 983
C.10.2. Вложенные пространства имен 984
C.10.3. Пространства имен и классы 985
C.11. Управление режимами доступа 985
C.11.1. Доступ к членам класса 985
C.11.2. Доступ к базовым классам 986
C.11.3. Доступ ко вложенным классам 988
C.11.4. Отношение «дружбы» 989
C.12. Указатели на члены классов 989
C.13. Шаблоны 990
C.13.1. Статические члены 990
C.13.2. Друзья 991
C.13.3. Шаблоны в качестве параметров шаблонов 992
C.13.4. Логический вывод аргументов функциональных шаблонов 992
C.13.5. Шаблоны и ключевое слово typename 993
C.13.6. Ключевое слово template в качестве квалификатора 995
C.13.7. Конкретизация 995
C.13.8. Связывание имен 996
C.13.8.1. Зависимые имена 997
C.13.8.2. Связывание в точке определения 999
C.13.8.3. Связывание в точке конкретизации 1000
C.13.8.4. Шаблоны и пространства имен 1001
C.13.9. Когда нужны специализации 1003
C.13.9.1. Конкретизация шаблона функции 1003
C.13.10. Явная конкретизация 1004
C.14. Советы 1005

Приложение D. Локализация 1007
D.1. Национальные особенности 1007
D.1.1. Программирование национальных особенностей 1008
D.2. Класс locale 1011
D.2.1. Локальные контексты с заданными именами 1013
D.2.1.1. Конструирование новых объектов локализации 1015
D.2.2. Копирование и сравнение констекстов локализации 1017
D.2.3. Контексты global() и classic() 1018
D.2.4. Сравнение строк 1019
D.3. Фасеты 1020
D.3.1. Доступ к фасетам класса locale 1021
D.3.2. Простой пользовательский фасет 1023
D.3.3. Использование локализаций и фасетов 1026
D.4. Стандартные фасеты 1026
D.4.1. Сравнение строк 1029
D.4.1.1. Именованные фасеты сравнения 1032
D.4.2. Ввод и вывод чисел 1033
D.4.2.1. Пунктуация чисел 1033

D.4.2.2. Вывод чисел . . . 1035
D.4.2.3. Ввод чисел . . . 1038
D.4.3. Ввод и вывод финансовой информации . . . 1039
D.4.3.1. Пунктуация денежных величин . . . 1040
D.4.3.2. Вывод денежных величин . . . 1043
D.4.3.3. Ввод денежных величин . . . 1044
D.4.4. Ввод и вывод дат и времени . . . 1046
D.4.4.1. Часы и таймеры . . . 1046
D.4.4.2. Класс Date . . . 1049
D.4.4.3. Вывод дат и времени . . . 1049
D.4.4.4. Ввод дат и времени . . . 1052
D.4.4.5. Более гибкий класс Date . . . 1054
D.4.4.6. Задание формата даты . . . 1056
D.4.4.7. Фасет ввода даты . . . 1058
D.4.5. Классификация символов . . . 1063
D.4.5.1. Вспомогательные шаблоны функций . . . 1066
D.4.6. Преобразование кодов символов . . . 1067
D.4.7. Сообщения . . . 1071
D.4.7.1. Использование сообщений из других фасетов . . . 1073
D.5. Советы . . . 1075
D.6. Упражнения . . . 1075

Приложение E. Исключения и безопасность стандартной библиотеки . . . 1077

E.1. Введение . . . 1077
E.2. Исключения и безопасность . . . 1079
C.3. Технологии реализации безопасности при исключениях . . . 1083
E.3.1. Простой вектор . . . 1083
E.3.2. Явное управление памятью . . . 1087
E.3.3. Присваивание . . . 1088
E.3.4. Метод push_back() . . . 1091
E.3.5. Конструкторы и инварианты . . . 1093
E.3.5.1. Применение функции init() . . . 1095
E.3.5.2. Полагаемся на действительное состояние по умолчанию . . . 1096
E.3.5.3. Отложенное выделение ресурсов . . . 1097
E.4. Гарантии стандартных контейнеров . . . 1098
E.4.1. Вставка и удаление элементов . . . 1099
E.4.2. Гарантии и компромиссы . . . 1102
E.4.3. Функция swap() . . . 1105
E.4.4. Инициализация и итераторы . . . 1106
E.4.5. Ссылки на элементы . . . 1106
E.4.6. Предикаты . . . 1107
E.5. Другие части стандартной библиотеки . . . 1108
E.5.1. Строки . . . 1108
E.5.2. Потоки . . . 1109
E.5.3. Алгоритмы . . . 1109
E.5.4. Типы valarray и complex . . . 1110
E.5.5. Стандартная библиотека языка C . . . 1110
E.6. Рекомендации пользователям стандартной библиотеки . . . 1110
E.7. Советы . . . 1113
E.8. Упражнения . . . 1114

Предметный указатель . . . 1117

Предисловие переводчика и редактора

Предыдущий перевод на русский язык книги Бьерна Страуструпа *Язык программирования C++* был выполнен в 2000 году. В течение десяти лет книга Страуструпа на русском языке неизменно пользовалась большим успехом у читателей. Книга от создателя языка C++ заслуженно считается классикой жанра во всем мире. Учитывая столь высокую роль книги, издательство Бином приняло решение о выполнении нового перевода.

Настоящий перевод 2010 года значительно отличается от перевода 2000 года в нескольких отношениях.

Во-первых, за счет объединения функций переводчика и редактора перевода, а также за счет того, что переводчик сам является автором большого печатного труда на ту же тему, удалось получить более гладкий текст на русском языке, ибо объективная сложность работы, которая стояла перед переводчиками и редакторами первого перевода на русский язык сказалась на некоторой тяжеловесности построения фраз, иногда затуманивающей исходный смысл высказываний автора.

Во-вторых, в рамках первого перевода было допущено несколько ошибок, выявить и исправить которые в переводе 2010 года не составило большого труда из-за близкого знакомства переводчика с данной предметной областью.

В-третьих, были учтены найденные за несколько лет ошибки самого автора — Бьерна Страуструпа, обнаруженные им и читателями и исправленные на сайте поддержки книги *http://www.research.att.com/~bs/3rd_errata.html.*

В-четвертых, за десять лет, прошедшие с момента выполнения первого русского перевода, постепенно изменялась и уточнялась система русской терминологии, причем процесс этот спонтанно отражался в учебной литературе, и его статистически среднее направление принято нами за основу при выполнении нового перевода.

В заключение, я надеюсь на то, что новый перевод позволит точнее донести содержание замечательной книги Бьерна Страуструпа до русскоязычных читателей. Свои замечания касательного перевода вы можете присылать редактору перевода 2010 года по адресу *info@binom-press.ru.*

Мартынов Н.Н.
10 февраля 2010 года, Москва

Предисловие автора к третьему русскому изданию

В августе 1998 года был ратифицирован стандарт языка C++ (ISO/IEC 14882 «Standard for the C++ Programming Language», результат голосования национальных комитетов по стандартизации: 22—0). Это событие знаменует новую эру стабильности и процветания C++, а также связанных с языком инструментальных средств и приемов программирования.

Лично для меня главным является то, что стандартный C++ лучше чем любая его предыдущая версия соответствует моему замыслу C++. Стандартный C++ и его стандартная библиотека позволяют мне писать лучшие, более элегантные и более эффективные программы, чем те, что я мог писать в прошлом.

Чтобы добиться этого, я и сотни других людей долго и интенсивно работали в комитетах по стандартизации ANSI и ISO. Целью этих усилий было описать язык и библиотеку, которые будут исправно служить всем пользователям языка, не предоставляя каких-либо преимуществ одной группе пользователей, компании или стране перед остальными. Процесс стандартизации был открыт, справедлив, нацелен на качество и согласие.

Усилия по стандартизации были инициированы Дмитрием Ленковым (Dmitry Lenkov) из Hewlett-Packard, который работал в качестве первого председателя комитета. Во время завершения работ над стандартом председателем был Стив Кламаг (Steve Clamage) из Sun. Джонатан Шапиро (Jonathan Shopiro) и Эндрю Кениг (Andrew Koenig) (оба из AT&T) были редакторами, которые — с помощью многих членов комитета — подготовили текст стандарта, основанный на моем исходном справочном руководстве по C++.

Открытый и демократичный процесс стандартизации потенциально таит в себе одну опасность: результат может оказаться «комитетской разработкой». В случае с C++ этого по большей части удалось избежать. Во-первых, я состоял председателем рабочей группы по расширению языка. В этом качестве я оценивал все основные предложения и составлял окончательные варианты тех из них, которые я, рабочая группа и комитет считали стоящими и в тоже время реализуемыми. Таким образом, комитет в основном обсуждал направляемые ему относительно завершенные проекты, а не занимался разработкой таковых. Во-вторых, главная из новых составляющих стандартной библиотеки — стандартная библиотека шаблонов STL, предоставляющая общую, эффективную, типобезопасную и расширяемую среду контейнеров, итераторов и алгоритмов — была, в основном, результатом работы одного человека, Александра Степанова (Alexander Stepanov) из Hewlett-Packard.

Важно, что стандарт C++ — это не просто бумажка. Он уже встроен в наиболее свежие реализации C++. Большинство основных реализаций поддерживают стандарт (с очень немногочисленными исключениями), причем все обещают полное соответствие в течение ближайших месяцев. Чтобы стимулировать честность производителей, две компании предлагают пакеты, проверяющие «стандартность» реализации C++.

Итак, код, который я пишу сейчас, использует, если это необходимо, большинство возможностей, предлагаемых стандартным C++ и описанных в данном третьем издании «Языка программирования C++».

Улучшения в языке C++ и расширение стандартной библиотеки заметно изменили то, как я пишу код. Мои теперешние программы короче, яснее и эффективнее прежних, что является прямым результатом лучшей, более ясной и систематичной поддержки абстракций в стандартном C++. Улучшенная поддержка таких средств, как шаблоны и исключения, сокращают потребность в более низкоуровневых и запутанных возможностях языка. Поэтому я считаю, что стандартный C++ проще в использовании, чем его предшествующие варианты, где мне приходилось имитировать средства, которые теперь поддерживаются непосредственно.

Основываясь на приведенных наблюдениях, я решил, что необходимо полностью переписать второе издание «Языка программирования C++». Никак иначе нельзя было должным образом отразить новые возможности, предлагаемые языком, и поддерживаемые ими приемы программирования и проектирования. В результате 80% материала третьего издания добавлено по сравнению со вторым. Изменилась и организация книги — теперь больший упор делается на технику и стиль программирования. Рассматриваемые изолированно, особенности языка скучны — они оживают лишь в контексте приемов программирования, ради поддержки которых и задумывались.

Соответственно, C++ может теперь преподаваться как язык более высокого уровне. То есть основное внимание можно уделять алгоритмам и контейнерам, а не жонглированию битами, объединениями, строками в стиле C, массивами и проч. Более низкоуровневые понятия (такие как массивы, нетривиальное использование указателей и приведения типов), естественно, также придется изучить. Но их представление теперь удается отложить до тех пор, пока новичок в программировании на C++, читатель или студент не обретут зрелость, необходимую, чтобы воспринимать эти средства в контексте более высокоуровневых концепций, к применению которых обучаемые уже привыкли.

В частности, невозможно переусердствовать в подчеркивании важности преимущества статически типобезопасных строк и контейнеров перед стилями программирования, предполагающими использование множества макросов и приведений типов. В третьем издании «Языка программирования C++» мне удалось избавиться от почти всех макросов и свести использование приведений типов к немногочисленным случаям, когда они действительно существенны. Я считаю серьезным недостатком принятую в C/C++ форму макросов, которую теперь можно считать устаревшей благодаря наличию более подходящих средств языка, таких как шаблоны, пространства имен, встроенные функции и константы. Точно также, широкое использование приведений типов в любом языке сигнализирует о плохом проектировании. Как макросы, так и приведения являются частыми источниками ошибок.

Тот факт, что без них можно обойтись, делает программирование на C++ куда более безопасным и элегантным.

Стандарт C++ предназначен для того, чтобы изменить наш способ программировать на C++, проектировать программы и обучать программированию на C++. Подобные изменения не происходят за один вечер. Я призываю вас не торопясь, внимательно присмотреться к стандарту C++, к приемам проектирования и программирования, используемым в третьем издании «Языка программирования C++», — и к вашему собственному стилю программирования. Предполагаю, что вполне возможны значительные улучшения. Но пусть ваша голова остается холодной! Чудес не бывает — при написании кода опасно использовать языковые возможности и приемы, которые вы понимаете лишь частично. Настало время изучать и экспериментировать — только поняв новые концепции и приемы, можно воспользоваться по-настоящему существенными преимуществами стандартного C++.

Добро пожаловать!
Бьерн Страуструп

Предисловие

Программирование есть понимание.
— Кристин Нюгард

Применение языка C++ доставляет мне все большее удовольствие. За прошедшие годы средства поддержки проектирования и программирования на C++ кардинально улучшились, а также было предложено множество полезных и удобных методик их использования. Но речь идет не только об удовольствии. Практикующие программисты достигли значительного повышения производительности, уровня поддержки, гибкости и качества в проектах любого уровня сложности и масштаба. К настоящему времени C++ оправдал большую часть возлагавшихся на него надежд, а кроме того он преуспел в решении задач, о которых я даже и не мечтал.

Эта книга представляет стандарт C++[1] и ключевые методы программирования и проектирования, поддерживаемые этим языком. Стандарт C++ намного мощнее и точнее, чем то, что излагалось в первом издании данной книги. Новые языковые черты, такие как пространства имен, исключения, шаблоны и механизм динамического определения типов, позволяют реализовывать многие методики программирования более непосредственно, чем это было возможно ранее. Стандартная библиотека предоставляет программистам возможность стартовать с существенно более высокоуровневых конструкций чем те, что доступны в «голом» языке.

Всего лишь треть информации из первого издания перекочевало во второе издание данной книги. Третье издание переработано еще более интенсивно. В ней есть кое-что, что будет полезно самым опытным программистам на C++. В то же время, она более доступна новичкам, чем предыдущие издания этой книги. Это стало возможным благодаря взрывному росту практического использования C++ и опыту, вытекающему из этой практики.

Стандартная библиотека позволяет по-новому представить основные концепции языка C++. Как и предыдущие издания, настоящая книга описывает язык C++ независимо от конкретных реализаций. Как и в предыдущих изданиях, все конструкции и концепции излагаются «снизу-вверх», так что к тому моменту, когда они начинают использоваться, они уже представлены и подробно описаны. Отметим, что начать использовать хорошо спроектированную библиотеку можно прежде, чем читатели смогут изучить ее внутреннее устройство. Кроме того, стандартная библиотека сама по себе является ценнейшим источником примеров программирования и методов проектирования.

В настоящей книге рассматриваются все основные средства языка и стандартной библиотеки. Ее материал организован вокруг этих основных средств, однако фокус изложения направлен на применение средств языка как инструмента проектирова-

---

[1] ISO/IEC 14882:2003(E), Стандарт языка программирования C++

ния и программирования, а не на формальные конструкции как таковые. Книга демонстрирует ключевые методики, делающие C++ эффективным, и излагает эти методики так, чтобы их постижение повышало мастерство программиста. За исключением глав, иллюстрирующих технические детали, практические примеры кода взяты из области системного программирования. Дополнением к данной книге может служить исчерпывающий справочник *The Annotated C++ Language Standard.*

Основной целью данной книги является стремление помочь читателю понять, как средства языка C++ непосредственно поддерживают ключевые методики программирования. Задача ставится так, чтобы подвигнуть читателя отойти как можно дальше от простого копирования примеров кода и эмулирования методик, присущих другим языкам программирования. Только отличное понимание идей, стоящих за средствами языка, ведет к настоящему мастерству. Дополненная деталями реализации информация из настоящей книги является достаточной для выполнения реальных проектов заметного масштаба. Я надеюсь, что эта книга поможет читателю по-новому взглянуть на программирование на языке C++ и стать более квалифицированным программистом и проектировщиком.

Благодарности

Помимо людей, упомянутых в соответствующих разделах предыдущих изданий книги, я хотел бы поблагодарить еще и следующих: Matt Austern, Hans Boehm, Don Caldwell, Lawrence Crowl, Alan Feuer, Andrew Forrest, David Gay, Tim Griffin, Peter Juhl, Brian Kernighan, Andrew Koenig, Mike Mowbray, Rob Murray, Lee Nackman, Joseph Newcomer, Alex Stepanov, David Vandevoorde, Peter Weinberger и Chris Van Wyk — за комментирование отдельных глав третьего издания. Без их помощи текст книги был бы менее понятен, содержал бы больше ошибок, был бы менее полным, а книга была бы тоньше.

Я также хотел бы поблагодарить добровольцев из Комитета по стандартизации C++ за их титанический труд, сделавший C++ тем, чем он сейчас является. Несправедливо здесь выделять кого-либо, но еще более несправедливо было бы не упомянуть следующих: Mike Ball, Dag Bruck, Sean Corfield, Ted Goldstein, Kim Knuttila, Andrew Koenig, Jose Lajoie, Dmitry Lenkov, Nathan Myers, Martin O'Riordan, Tom Plum, Jonathan Shopiro, John Spicer, Jerry Schwarz, Alex Stepanov и Mike Vilot — каждый из них работал со мной над отдельными частями C++ или стандартной библиотеки.

После выхода первого тиража этой книги десятки людей присылали мне исправления и предложения по ее улучшению. Многие из этих предложений я учел при подготовке более поздних тиражей, от чего они только выиграли. Переводчики этой книги на иностранные языки также высказывали свои соображения относительно пояснения отдельных моментов. Отвечая на просьбы читателей я добавил приложения D и E. Особо хотел бы при этом поблагодарить следующих людей: Dave Abrahams, Matt Austern, Jan Bielawski, Janina Mincer Daszkiewicz, Andrew Koenig, Dietmar Kuhl, Nicolai Josuttis, Nathan Myers, Paul E. Sevin, Andy Tenne-Sens, Shoichi Uchida, Ping-Fai (Mike) Yang и Dennis Yelle.

Мюррей Хилл, Нью Джерси *Бьерн Страуструп*

Предисловие ко второму изданию

А дорога бежит и бежит.
— Бильбо Бэггинс

Как было предсказано в первом издании этой книги, C++ эволюционировал в соответствии с запросами его пользователей. Эта эволюция направлялась запросами пользователей самой разной квалификации, работающих в разных предметных областях. Сообщество программистов на C++ выросло в сотни раз за шесть лет, прошедшие с момента выхода первого издания этой книги. За это время были разработаны многочисленные методики, усвоены полезные практические уроки, приобретен бесценный опыт. Часть этого опыта отражена в настоящем издании.

Главной целью развития и расширения языка за эти шесть лет было стремление сделать C++ языком абстракции данных и языком объектно-ориентированного программирования в целом, языком, пригодным для написания высококачественных библиотек, в том числе библиотек пользовательских типов. Высококачественная библиотека — это библиотека, которая предоставляет пользователю концепцию в виде одного или нескольких классов, которые удобны в применении, безопасны и эффективны Здесь безопасность означает предоставление пользователю интерфейса к содержимому библиотеки, безопасному в отношении типов. Эффективность означает, что классовая структура библиотеки не налагает дополнительной нагрузки на процессор и память по сравнению с ручным кодом, написанным на C.

В настоящей книге приводится полное описание языка C++. В главах 1-10 сосредоточен вводный учебный материал. Главы 11-13 посвящены обсуждению вопросов проектирования и реализации; наконец, далее приведено полное справочное руководство по C++. Естественно, в настоящей книге присутствуют новые черты и свойства языка, появившиеся после выхода первого издания книги: уточненный механизм разрешения перегрузки, средства управления памятью, безопасный по типам механизм компоновки, константные и статические члены классов, абстрактные классы, множественное наследование, шаблоны, обработка исключений.

C++ является языком общего назначения; его естественной сферой применения является системное программирование в самом общем смысле этого термина. Тем не менее, C++ с успехом используется и в других предметных областях. Реализации C++ имеются и для маломощных микрокомпьютеров, и для мощнейших суперкомпьютерных систем и разных операционных сред. Поэтому настоящая книга описывает собственно язык C++ без его привязки к конкретным операционным системам и их библиотекам.

В книге имеется большое число примеров классов, которые будучи полезными должны быть охарактеризованы как «игрушечные». Такой стиль подачи общих принципов и полезных методик позволяет лучше сфокусировать на них внимание, чем это возможно в контексте больших реальных программ, где они были бы похоронены в многочисленных деталях. Большую часть полезных классов, приведенных в книге, таких как строки, связанные списки, массивы, матрицы, графические классы, ассоциативные массивы и т.д. можно найти в более совершенном виде в разных коммерческих и некоммерческих источниках. Многие из этих промышленных классов являются прямыми или косвенными потомками приведенных здесь игрушечных классов.

По сравнению с предыдущим изданием настоящее больше сфокусировано на обучении, но форма подачи материала по-прежнему ориентирована на опытных программистов и разработчиков, дабы не оскорблять их опыт и квалификацию. Более широкое обсуждение вопросов проектирования отражает спрос на информацию, не замыкающуюся лишь на изучении средств языка и их непосредственного использования. Увеличено число приведенных технических деталей, а также качество их изложения. В частности, справочное руководство по языку отражает многолетний опыт работы в этом направлении. Мне хотелось написать книгу, которую к которой большинство программистов обращалось бы не один раз. Таким образом, данная книга представляет язык C++, его фундаментальные принципы и ключевые технологии, в рамках которых язык используется. Успехов!

Благодарности

Помимо людей, упомянутых в соответствующих разделах первого издания книги, я хотел бы поблагодарить еще и следующих: Al Aho, Steve Buroff, Jim Coplien, Ted Goldstein, Tony Hansen, Lorraine Juhl, Peter Juhl, Brian Kernighan, Andrew Koenig, Bill Leggett, Warren Montgomery, Mike Mowbray, Rob Murray, Jonathan Shopiro, Mike Vilot, and Peter Weinberger — за комментирование отдельных глав второго издания.

Многие люди оказали влияние на развитие C++ с 1985 года по 1991 год. Я могу упомянуть лишь некоторых из них: Andrew Koenig, Brian Kernighan, Doug McIlroy и Jonathan Shopiro. Также большое спасибо всем участникам просмотра и рецензирования черновика справочного руководства и всем людям, настрадавшимся в первый год работы комитета X3J16.

Мюррей Хилл, Нью Джерси *Бьерн Страуструп*

Предисловие к первому изданию

Язык формирует наше мышление и определяет то, о чем мы можем думать.
— Б. Л. Ворф

C++ является языком общего назначения, цель которого — сделать работу серьезного программиста более приятной. За исключением некоторых деталей, C++ является надмножеством языка программирования C. Помимо возможностей языка C, C++ предоставляет гибкие и эффективные средства для определения новых типов. Программист может сегментировать приложение на фрагменты, определив для этого новые типы, отвечающие концепциям приложения. Такую технику программирования часто называют абстракцией данных. Объекты пользовательских типов содержат информацию, характерную для этих типов. Такие объекты можно с удобством и безопасностью использовать в контекстах, когда их тип неизвестен на этапе компиляции. Программы, использующие подобного рода объекты, часто называют объектными. При надлежащем применении такая техника ведет к более коротким программам, более понятным и более удобным в сопровождении.

Ключевой концепцией языка C++ является класс. Класс — это тип, определяемый пользователем. Классы обеспечивают сокрытие данных, гарантированную инициализацию данных, неявное преобразование между пользовательскими типами, динамическое определение типа, контролируемое управление памятью и механизмы перегрузки операций. C++ гораздо лучше контролирует типы, чем C, а также позволяет достичь более высокой модульности. Он также содержит улучшения, не имеющие прямого отношения к классам, такие как символические константы, встраиваемые функции, аргументы функций по умолчанию, перегрузка имен функций, операции по управлению выделением памяти, а также ссылки. C++ сохраняет возможности языка C по эффективной работе с низкоуровневыми аппаратнозависимыми базовыми типами (битами, байтами, словами, адресами и т.д.). Это позволяет реализовать пользовательские типы с высокой эффективностью.

C++ и его стандартные библиотеки построены с учетом переносимости. Текущая реализация будет работать на большинстве систем, поддерживающих язык C. Библиотеки языка C можно использовать в программах на C++, также как и большинство инструментальных средств языка C.

Данная книга предназначена в первую очередь для того, чтобы серьезные программисты могли изучить C++ и использовать его в нетривиальных разработках. Книга содержит полное описание C++, множество законченных примеров и еще большее количество фрагментов программ.

Благодарности

C++ никогда не достиг бы необходимой зрелости без интенсивного использования, советов и конструктивной критики со стороны друзей и коллег. В частности, Tom Cargill, Jim Coplien, Stu Feldman, Sandy Fraser, Steve Johnson, Brian Kernighan, Bart Locanthi, Doug McIlroy, Dennis Ritchie, Larry Rosler, Jerry Schwarz и Jon Shopiro подали важные для развития языка идеи. Dave Presotto написал текущую реализацию библиотеки потокового ввода/вывода.

Кроме того, сотни людей способствовали развитию языка C++ и его компилятора, присылая мне свои предложения по улучшениям, описания проблем, с которыми они сталкивались, а также сообщения об ошибках компилятора. Я могу отметить здесь лишь некоторых: Gary Bishop, Andrew Hume, Tom Karzes, Victor Milenkovic, Rob Murray, Leonie Rose, Brian Schmult, и Gary Walker.

Множество людей помогали с изданием данной книги. Среди них: Jon Bentley, Laura Eaves, Brian Kernighan, Ted Kowalski, Steve Mahaney, Jon Shopiro и участники семинара по C++ в Bell Labs, Columbus, штат Ohio (Огайо), 26-27 июня 1985 года.

Мюррей Хилл, Нью Джерси — *Бьерн Страуструп*

Введение

Во введении представлен обзор основных концепций и свойств языка программирования C++ и его стандартной библиотеки. Также поясняется структура книги и подход к изложению средств языка и методов их применения. Кроме того, во вводных главах дана базовая информация о языке C++, архитектуре и практике использования.

Главы

1. Обращение к читателю
2. Обзор языка C++
3. Обзор стандартной библиотеки

«... и ты, Маркус, ты дал мне многое. Теперь я дам тебе хороший совет. Будь многими. Брось играть в бытие одним лишь Маркусом Кокоза. Ты так сильно волновался за Маркуса Кокоза, что стал его рабом и пленником. Ты не делал ничего, не подумав сначала о его благополучии и престиже. Ты всегда боялся того, что Маркус совершит какую-либо глупость или ему станет скучно. А что в этом страшного? Во всем мире люди совершают глупости. Я хочу, чтобы тыл жил легче. Будь больше, чем одним, будь многими людьми, столь многими, сколь можешь представить себе.»

— Карен Бликсен (Karen Blixen)

(«The Dreamers» из книги «Seven Gothic Tales», написанной под псевдонимом Isak Dinesen, Random House, Inc © Isak Dinesen, 1934, обновлено в 1961)

1

Обращение к читателю

И молвил Морж:
«Пришла пора поговорить о многом».
— Льюис Кэрролл

https://t.me/it_boooks

Структура этой книги — как изучать C++ — структура C++ — эффективность и структура — философские замечания — исторические замечания — для чего используется C++ — C и C++ — рекомендации для программистов на C — рекомендации для программистов на C++ — размышления о программировании на C++ — советы — ссылки.

1.1. Структура книги

Книга состоит из шести частей:

- Введение: В главах 1–3 дается обзор языка C++, поддерживаемых им стилей программирования и стандартной библиотеки C++.
- Часть I: В главах 4–9 изучаются встроенные типы языка C++ и базовые средства построения программ.
- Часть II: Главы 10–15 содержат учебный материал по объектно-ориентированному и обобщенному программированию на C++.
- Часть III: В главах 16–22 представлена стандартная библиотека языка C++.
- Часть IV: В главах 23–25 рассматриваются проблемы, связанные с проектированием и разработкой программ.
- Приложения: Приложения A–E содержат технические детали языка C++.

В главе 1 дается обзор книги, рекомендации по ее чтению, а также общие сведения о языке и способах его применения. Вы можете бегло ознакомиться с ее содержимым, задержавшись лишь на интересных местах, а позднее вернуться к ней снова, после прочтения других частей данной книги.

Главы 2 и 3 содержат обзор основных концепций и свойств языка C++ и его стандартной библиотеки. Цель этих глав — обратить ваше внимание на важность

понимания фундаментальных концепций и базовых свойств языка, демонстрируя то, что может быть выражено полным набором языковых средств. По крайней мере, эти главы должны убедить читателя в том, что язык C++ — это не C, и что он прошел большой путь со времени первого и второго изданий этой книги. Глава 2 посвящена знакомству с высокоуровневыми чертами языка C++: поддержкой абстракции данных, объектно-ориентированным и обобщенным программированием. В главе 3 представлены базовые принципы и основные средства стандартной библиотеки. Это позволит мне применять стандартную библиотеку в последующих главах, а вам — использовать в упражнениях, вместо того, чтобы полагаться исключительно на встроенные низкоуровневые средства языка.

Вводные главы демонстрируют основной способ представления материала, принятый в данной книге: для того, чтобы обеспечить предметное и реалистичное изучение конкретного вопроса, я вначале коротко излагаю лишь основы концепции, а подробное и углубленное рассмотрение выполняю позже. Такой подход позволяет мне обращаться к конкретным примерам до того, как будут изучены все детали. Можно сказать, организация книги соответствует положению, что мы лучше обучаемся, если продвигаемся от конкретного к абстрактному — даже там, где абстрактные идеи кажутся в ретроспективе простыми и очевидными.

В части I рассматривается подмножество C++, поддерживающее стили программирования, характерные для языков C или Pascal. Эта часть книги охватывает фундаментальные типы, выражения и управляющие конструкции программ на C++. Модульность в той части, что поддерживается пространствами имен, исходными файлами и обработкой исключений, также рассматривается. Я предполагаю, что вы знакомы с фундаментальными основами программирования, излагаемыми в части I. Поэтому, например, хоть я и рассматриваю рекурсию и итерацию в этой части книги, я не трачу много времени на прояснение вопроса о реальной пользе этих концепций.

Часть II посвящена средствам C++ для определения и использования новых типов. Здесь представлены (глава 10, глава 12) конкретные и абстрактные классы (интерфейсы), а также перегрузка операций (глава 11), полиморфизм и иерархии классов (глава 12, глава 15). В главе 13 представлены шаблоны, то есть средства для определения семейств типов и функций. Она также демонстрирует способы создания контейнеров (например, списков) и приемы обобщенного программирования. В главе 14 рассматриваются обработка исключений, методы обработки ошибок и общая стратегия создания устойчивых и надежных программ. Я предполагаю, что вы или не слишком хорошо знакомы с объектно-ориентированным и обобщенным программированием, или вам не хватает подробных объяснений того, как именно эти базовые абстракции поддерживаются средствами языка C++. Поэтому я рассматриваю не только сами абстракции, но также и технику их применения. В части IV эта тема развивается далее.

Часть III посвящена стандартной библиотеке языка C++. В ней объясняется, как пользоваться библиотекой, каков ее дизайн и техника применения, а также показано, как расширить ее возможности. Библиотека предоставляет контейнеры (такие как ***list***, ***vector*** и ***map***; глава 16, глава 17), стандартные алгоритмы (такие как ***sort***, ***find*** и ***merge***; глава 18, глава 19), строки (глава 20), ввод/вывод (глава 21) и поддержку вычислений (глава 22).

Часть IV затрагивает вопросы, актуальные для ситуаций, когда C++ используется в проектировании и реализации больших программных систем. В главе 23 рассматрива-

ются вопросы проектирования и управления ходом выполнения проектов. В главе 24 обсуждается связь между языком программирования и приемами проектирования. Глава 25 демонстрирует некоторые способы применения классов в проектировании.

Приложение А содержит грамматику C++ с некоторыми комментариями. В приложении В обсуждается связь между C++ и C, а также между стандартным C++ (называемым также ISO C++ или ANSI C++) и предыдущими его версиями. В приложении С представлены некоторые технические детали языка C++. В приложении D рассматриваются средства стандартной библиотеки, предназначенные для локализации программного обеспечения. В приложении Е обсуждаются вопросы гарантии возбуждения исключений и соответствующие требования стандартной библиотеки.

1.1.1. Примеры и ссылки

В данной книге акцент сделан скорее на организации программ, а не на реализации конкретных алгоритмов. Как следствие, я избегаю «слишком умных» или просто трудных для понимания алгоритмов. Тривиальный алгоритм лучше подходит для иллюстрации языковых элементов или структуры программы. Например, я использую сортировку Шелла, тогда как в реальном коде лучше применить быструю сортировку. Часто в качестве упражнений предлагается переработать учебный код под более подходящий алгоритм. И, наконец, в реальной практике лучше использовать библиотечные функции вместо приведенного в книге иллюстративного кода.

Учебные примеры неизбежно искажают представление о разработке программ, так как упрощение кода примеров одновременно приводит к исчезновению сложностей, связанных с масштабов проблем. Я не представляю другого способа почувствовать, что такое реальное программирование и каков в действительности язык программирования, иначе как в процессе практической реализации программных систем «настоящего размера». В данной книге основное внимание уделяется элементам языка и технике программирования, на основе которых и строятся все программы.

Отбор примеров отражает мой собственный опыт разработки компиляторов, базовых библиотек и программ-симуляторов (моделирование; программы-тренажеры). Примеры являются упрощенными вариантами того, что встречается в реальной практике. Упрощение необходимо, иначе язык программирования и вопросы проектирования затеряются во множестве мелких деталей. В книге нет «особо умных» примеров, не имеющих аналогов в реальных программах. Везде, где только можно, я отношу в приложение С иллюстрирующие технические детали языка примеры, которые используют идентификаторы ***x*** или ***y*** для переменных, имена ***A*** или ***B*** для типов, и обозначения ***f***() и ***g***() для функций.

Везде, где возможно, язык и библиотека рассматриваются скорее в контексте их применения, нежели в стиле «сухого справочника». Представленные в книге языковые элементы и степень детальности их описания отражают мои личные представления о том, что действительно нужно для эффективного программирования на C++. Другая моя книга (в соавторстве с Э. Кенигом — Andrew Koenig) — The Annotated C++ Language Standard, служит дополнением к настоящей книге и содержит полное и всестороннее определение языка C++, а также поясняющие комментарии. Логичным было бы еще одно дополнение — The Annotated C++ Standard Library. Но ввиду ограниченности сил и времени я не могу обещать выход такой книги.

Ссылки на разделы данной книги даются в форме §2.3.4 (глава 2, раздел 3, подраздел 4), §B.5.6 (приложение В, подраздел 5.6) и §6.6[10] (глава 6, упражнение 10).

1.1.2. Упражнения

Упражнения размещены в концах глав данной книги. По большей части они формулируются в стиле «напиши программу». Всегда пишите программы, пригодные для успешной компиляции и нескольких тестовых прогонов. Упражнения существенно разнятся по уровню сложности, так что для каждого из них приводится приблизительная условная оценка. Масштаб оценок экспоненциальный, так что если, например, упражнение, оцененное как (*1), занимает 10 минут, то упражнение с оценкой (*2) может занять час, а (*3) — целый день. Кроме того, время, необходимое для выполнения упражнения, зависит больше от вашего опыта, чем от самого упражнения. Например, упражнение с оценкой (*1) может занять и целый день, если вы при этом параллельно знакомитесь с компьютерной системой и выясняете детали работы с ней. А с другой стороны, упражнение с оценкой (*5) может быть решено кем-нибудь за один час, если у него под рукой набор готовых программ и утилит.

Любую книгу по языку С можно использовать в качестве источника дополнительных упражнений к части I. А любая книга по структурам данных и алгоритмам послужит источником упражнений для частей II и III.

1.1.3. Замечания о конкретных реализациях языка (компиляторах)

В книге используется «чистый C++», как он определен в стандарте [C++, 1998], [C++, 2003]. Поэтому примеры из книги должны компилироваться в любых стандартных реализациях. Большинство из них были проверены на нескольких компиляторах. При этом не все компиляторы справились с новейшими элементами языка C++. Однако я не думаю, что есть необходимость в том, чтобы явно называть такие реализации, ибо разработчики непрерывно работают над их улучшением, и эта информация быстро устареет. В приложении В приведены советы по устранению проблем со старыми компиляторами, а также с кодом, написанным для компиляторов языка С.

1.2. Как изучать C++

Изучая C++, нужно сконцентрироваться на концепциях и не потеряться в технических деталях. Цель изучения — стать квалифицированным программистом, который эффективно проектирует и реализует новые разработки, а также умело поддерживает уже существующие. Для этого важнее понять методологию проектирования и программирования, чем исчерпывающе изучить все детали языка — последнее постепенно приходит вместе с опытом.

C++ поддерживает множество стилей программирования. Все они основаны на строгой статической проверке типов, и большинство из них нацелены на достижение высокого уровня абстракции и непосредственного представления идей программиста. При этом каждый из стилей достигает свою цель без снижения эффективности выполнения программы и заметного увеличения потребляемых компью-

терных ресурсов. Программисты, переходящие с иных языков (например, С, Fortran, Smalltalk, Lisp, ML, Ada, Eiffel, Pascal, Modula-2) должны четко осознать: чтобы реально воспользоваться преимуществами языка C++, нужно потратить время, изучить и на практике овладеть стилями и приемами программирования, характерными для C++. То же относится и к программистам, привыкшим к работе с ранними, менее выразительными версиями C++.

Бездумный перенос на новый язык идей и методов программирования, доказавших свою эффективность для другого языка, в типичном случае приводит к неуклюжему, медленному и сложному в сопровождении коду. К тому же писать такой код крайне утомительно, ибо каждая его строчка и каждое сообщение компилятора об ошибке подчеркивают отличие применяемого языка от «старого». Вы можете программировать в стиле Fortran, C, Smalltalk и т.п. на любом языке, но делать это для языка с иной философией и неэкономно, и неприятно. Любой язык является щедрым источником идей о том, как следует писать программы на C++. Однако чтобы эти идеи стали эффективными в новом контексте, их нужно преобразовать в нечто, что хорошо согласуется с общей структурой и системой типов C++. Над базовой системой типов языка программирования возможна лишь пиррова победа.

C++ поддерживает постепенный подход к обучению. Ваш собственный путь изучения нового языка программирования зависит от того, что вы уже знаете, и чему именно хотите научиться. В этом деле нет единого подхода для всех и каждого. Я предполагаю, что вы изучаете C++, чтобы повысить эффективность в проектировании программ и их реализации. То есть я полагаю, что вы не только хотите изучить новый синтаксис, чтобы с ним писать новые программы старыми методами, но в первую очередь стремитесь научиться новым, более продуктивным способам построения программных систем. Это надо делать постепенно, так как приобретение и закрепление новых навыков требует времени и практики. Подумайте, сколько времени обычно занимает качественное овладение иностранным языком, или обучение игре на музыкальном инструменте. Конечно, стать квалифицированным программным архитектором и разработчиком можно быстрее, но не настолько, как многим хотелось бы.

Отсюда следует, что вы будете использовать C++ (зачастую для построения реальных систем) еще до того, как овладеете всеми его нюансами. Поддерживая несколько парадигм программирования (глава 2), C++ обеспечивает возможность продуктивной работы программистов разной квалификации. Каждый новый стиль программирования добавляет новый штрих в ваш совокупный инструментарий, но каждый стиль ценен и сам по себе, так как увеличивает эффективность вашей работы как программиста. Язык C++ организован таким образом, что вы можете изучать его концепции последовательно, постепенно наращивая свои возможности. Это важно, так как ваши возможности будут расти практически пропорционально затраченным на обучение усилиям.

В непрекращающихся дебатах по поводу необходимости (или отсутствия необходимости) предварительного изучения языка С я решительно на стороне тех, кто считает, что лучше сразу приступать к изучению C++: он безопаснее, он более выразителен и он снижает необходимость в низкоуровневом программировании. Легче понять хитроумные «штучки» на С, компенсирующие отсутствие в нем высокоуровневых конструкций, если предварительно ознакомиться с общими возможностями этих языков и некоторыми высокоуровневыми средствами C++.

Приложение B поможет программистам при переходе от C++ к C, например, с целью поддержки ранее созданного кода.

В настоящее время существует несколько независимо разработанных реализаций C++. Масса инструментов, библиотек, графических сред разработки к вашим услугам. Великое множество учебников, руководств, журналов, бюллетеней, электронных досок объявлений, почтовых рассылок, конференций и курсов призваны снабдить вас самой последней информацией о новых реализациях C++, их инструментах, библиотеках и т.д. Если вы планируете использовать C++ самым серьезным образом, я настоятельно рекомендую получить доступ к указанным источникам информации. Так как у каждого из них свой взгляд на C++, то целесообразно обратиться хотя бы к двум таким источникам. Например, см. [Barton, 1994], [Booch, 1994], [Henricson, 1997], [Koenig, 1997], [Martin, 1995].

1.3. Как проектировался C++

Простота была важным критерием при разработке C++ — там, где стоял выбор между простотой языковых конструкций и простотой реализации компилятора, выбиралось первое. Большое внимание уделялось совместимости с языком C [Koenig, 1989], [Stroustrup, 1994], (приложение B), что, однако, помешало «почистить» синтаксис языка C.

В языке C++ нет встроенных высокоуровневых типов и операций. Например, в C++ нет встроенных матриц с операциями их обращения, нет и строк с операциями конкатенации. Если такие типы нужны, то они могут быть созданы средствами языка. Более того, создание типов общего назначения или типов, специфичных для конкретной программы, и есть основная форма деятельности при программировании на C++. Хорошо спроектированный пользовательский тип отличается от встроенного типа только тем, как он определен, а не тем, как он используется. Описанная в части III стандартная библиотека содержит массу примеров таких типов данных. Для пользователя нет большой разницы между встроенными типами и типами стандартной библиотеки.

При проектировании C++ средства, которые замедляют работу программы и требуют дополнительных затрат памяти во всех случаях (даже когда их напрямую не используют), отвергались. Например, не применяются дополнительные внутренние конструкции, требующие индивидуального хранения вместе с каждым объектом. Таким образом, если, например, объявляется структура с двумя 16-битовыми величинами, то объекты этого типа целиком помещаются в 32-битовый регистр.

C++ разрабатывался для той же самой традиционной среды компиляции и исполнения, что была присуща языку C и операционной системы UNIX. Разумеется, C++ не ограничивается операционной системой UNIX; ему просто присущи те же самые модельные взаимоотношения между языком, библиотеками, компиляторами, компоновщиками, средой исполнения и т.д., что свойственны языку C и UNIX. Эта минимальная модель помогла языку C++ успешно работать практически на любой платформе. Тем не менее, целесообразно использовать C++ и в средах с существенно большей поддержкой. Динамическая загрузка, инкрементная компиляция, базы данных определений типов — все это можно с успехом использовать без обратного влияния на сам язык.

В C++ проверка типов и сокрытие данных полагаются на анализ программы во время компиляции, который позволяет в результате избегать непреднамеренной порчи данных. Эти меры не обеспечивают секретности или защиты от того, кто намеренно нарушает правила. Однако их можно использовать свободно, не опасаясь за эффективность программы. Главная идея состоит в том, что свойства языка должны быть не только элегантными, но и приемлемыми для реальных программ.

Систематическое и подробное описание структуры языка C++ см. в [Stroustrup, 1994].

1.3.1. Эффективность и структура

C++ разрабатывался как расширение языка C и за малыми исключениями содержит C в качестве подмножества. Это подмножество, являясь базовой частью языка, демонстрирует тесную связь между его типами, операциями, операторами и объектами, с которыми компьютер работает непосредственно: числами, символами и адресами. В языке C++ все выражения и операторы, за исключением операций new, delete, typeid, dynamic_cast, throw и try-блока, не требуют никакой дополнительной поддержки со стороны среды выполнения.

C++ использует те же самые схемы вызовов функций и последовательности возвратов, что и язык C — или более эффективные. Когда даже столь эффективные механизмы вызовов слишком накладны, C++ обеспечивает возможность прямой подстановки кода функций в точку вызова, так что удается совместить удобства функциональной декомпозиции и нулевые накладные расходы на функциональные вызовы.

Одной из целей разработки языка C было намерение устранить необходимость в ассемблерном кодировании наиболее требовательных системных задач. При разработке C++ мы постарались по максимуму сохранить это преимущество. Различия между C и C++ заключаются преимущественно в разной степени акцента на типах и структуре. C — выразителен и гибок. C++ — еще более выразителен. Однако чтобы эту выразительность реализовать в собственном коде, требуется уделять большее внимание типам объектов. Зная типы, компилятор корректно обращается с выражениями, тогда как в противном случае вам самим пришлось бы скрупулезно, в мельчайших деталях, определять каждую операцию. Зная типы, компилятор может обнаруживать такие ошибки, которые в противном случае оставались бы в программе до стадии тестирования — или даже дольше. Заметим, что применение системы типов для проверки аргументов вызова функций, для защиты данных от случайного изменения, для предоставления новых типов, для определения операций над новыми типами и т.п. не приводит к дополнительным накладным расходам во время выполнения программы.

Другой акцент C++ — акцент на структуре программы, отражает тот факт, что после разработки языка C произошел резкий рост среднего размера разрабатываемых программ. Небольшую программу (скажем, не более 1000 строк исходного кода) можно «силовым образом» заставить правильно работать, даже нарушая все правила хорошего стиля написания программ. Для больших программ таких простых решений уже нет. Если структура программы из 100000 строк запутана, то вы столкнетесь с ситуацией, когда новые ошибки возникают с той же скоростью, с какой устраняются старые. C++ разрабатывался так, чтобы обеспечить лучшую

структуру для больших программ и чтобы программист в одиночку мог справиться с большими объемами кода. Кроме того, хотелось, чтобы строка кода на C++ отражала больший объем работы, чем среднестатистическая строка кода на C или Pascal. К настоящему времени C++ перевыполнил поставленные перед ним задачи.

Не каждый фрагмент кода может быть хорошо структурирован, независим от аппаратуры, легко читаем и т.д. В C++ предусмотрены средства для прямого и непосредственного манипулирования аппаратными ресурсами без какого-либо намека на простоту восприятия и безопасность работы. Но такие средства всегда можно аккуратно спрятать за элегантными и безопасными интерфейсами.

Естественно, что применение C++ для написания больших программ сопровождается вовлечением в процесс многих программистов. Здесь-то и приносят дополнительные плоды гибкость языка C++ и его акцент на модульность и строго типизированные интерфейсы. Язык C++ имеет столь же сбалансированный набор вспомогательных средств для написания больших программ, что и многие другие языки. Но дело здесь в том, что по мере увеличения размера программ сложности, связанные с их разработкой и сопровождением, перетекают из области чисто языковых проблем в область создания удобного рабочего инструментария, графических сред разработки и управления проектами. Эти вопросы обсуждаются в части IV.

Настоящая книга акцентирует внимание на методах написании полезных программных средств, общеупотребительных типов, библиотек и т.п. Все это можно использовать и при написании небольших программ, и при разработке больших систем. Более того, поскольку все нетривиальные программы состоят из полунезависимых частей, то методы написания таких частей будут полезны всем программистам.

Наверное, вы полагаете, что написание программ посредством детального описания структуры используемых в ней типов приводит к увеличению совокупного размера исходного кода. Для языка C++ это не так. Программа на C++, которая определяет классы, точно определяет типы функциональных аргументов и т.д., немного короче эквивалентной, но не применяющей этих средств программы на C. Когда же используются библиотеки, программа на C++ уже намного короче своего C-эквивалента (если он вообще может быть создан).

1.3.2. Философские замечания

Язык программирования служит двум целям: он предоставляет программисту средства формулирования подлежащих выполнению действий, и он предоставляет набор концепций, которые программист использует, обдумывая, что нужно сделать. Первая цель в идеале требует языка, «близкого к машине», чтобы все машинные концепции отображались просто и эффективно способом, достаточно очевидным для программиста. Язык C был создан с этой целью. Вторая цель в идеале требует языка, близкого к решаемой задаче, чтобы понятия из логического пространства задачи могли формулироваться в терминах языка программирования кратко и непосредственно. Эту цель и преследуют средства, добавленные к языку C в процессе разработки C++.

Существует очень тесная связь между языком, на котором мы думаем/программируем, и задачами/решениями, которые мы можем себе представить. По этой причине ограничение языка средствами, не позволяющими программисту совер-

шать ошибки, по крайней мере неразумно. Как и в случае естественных языков, знание по крайней мере двух языков чрезвычайно полезно. Язык снабжает программиста набором концептуальных средств, которые, будучи неадекватными задаче, просто игнорируются. Хороший программный дизайн и отсутствие ошибок не могут гарантироваться всего лишь присутствием/отсутствием каких-либо языковых элементов.

Система типов особо полезна в случае нетривиальных задач. Для них концепция классов языка C++ на деле доказала свою чрезвычайную силу.

1.4. Исторические замечания

Я придумал язык C++, зафиксировал связанные с ним определения и выполнил самую первую реализацию. Также я выбрал и сформулировал критерии проектирования языка, разработал все основные элементы и рассматривал предложения по его расширению в комитете по стандартизации C++.

Ясно, что C++ многое заимствует у C [Kernighan, 1978]. За исключением устраненных лазеек в системе типов (см. приложение B), C остается подмножеством C++. Я сохранил все низкоуровневые средства языка C, чтобы можно было справляться с самыми критичными системными задачами. В свою очередь, C взял многое у своего предшественника — языка BCPL [Richards, 1980]; язык C++ позаимствовал у BCPL стиль однострочных комментариев // comment. Другим источником вдохновения был для меня язык Simula67 [Dahl, 1970], [Dahl, 1972]; концепция классов (вместе с производными классами и виртуальными функциями) была позаимствована у этого языка. Возможность перегружать операции и помещать объявления всюду, где может стоять оператор, напоминает Algol68 [Woodward, 1974].

С момента выхода первого издания этой книги язык интенсивно пересматривался и обновлялся. Основной ревизии подверглись правила разрешения перегрузки, компоновка и средства управления памятью. Дополнительно, были внесены небольшие изменения для улучшения совместимости с языком C. Также был сделан ряд обобщений и несколько фундаментальных дополнений: множественное наследование, статические функции-члены классов, константные функции-члены, защищенные члены классов, шаблоны, обработка исключений, RTTI (run-time type identification) и пространства имен (namespaces). Общим мотивом всех этих изменений и расширений было стремление улучшить средства языка для построения и использования библиотек. Эволюция C++ описана в [Stroustrup, 1994].

Шаблоны планировались, главным образом, для статической типизации контейнеров (таких, как списки, вектора и ассоциативные массивы) и для их элегантного и эффективного использования (обобщенное программирование). Ключевая цель заключалась в уменьшении применения макросов и операций явного приведения типов. На разработку шаблонов оказали влияние механизм обобщений языка Ada (своими достоинствами и недостатками) и параметризованные модули языка Clu. Аналогично, механизм исключений C++ испытал влияние со стороны языков Ada [Ichbiah, 1979], Clu [Liskov, 1979] и ML [Wikstrom, 1987]. Остальные нововведения языка C++ в период с 1985 по 1995 годы — множественное наследование, чисто виртуальные функции и пространства имен — проистекали главным образом из опыта применения C++, а не заимствовались из других языков.

Самые ранние версии языка, широко известные как «С с классами» [Stroustrup, 1994], использовались, начиная с 1980 года. Толчком к созданию языка послужило мое желание писать управляемые событиями программы-симуляторы, для которых язык Simula67 был бы идеальным, если бы не проблемы с эффективностью. «С с классами» использовался преимущественно в проектах, на которых это средство разработки особо эффективных (быстродействующих и требующих минимума памяти) программ проходило суровую обкатку и тестирование. В нем тогда отсутствовали перегрузка операций, ссылки, виртуальные функции, шаблоны, исключения и многие другие элементы. Первый выход C++ за стены исследовательской организации произошел в июле 1983 года.

Название C++ (читается «си плюс плюс») было придумано Риком Маскитти (Rick Mascitti) летом 1983 года. Это название отражает эволюцию языка со стороны С; в этом языке «++» означает операцию инкремента. Более короткое «С+» соответствует ошибочному выражению в языке С, и, к тому же, оно уже было занято (для имени другого языка). Язык не был назван D, поскольку он является прямым расширением С и не стремится «лечить» его проблемы отбрасыванием отдельных элементов. Иные интерпретации названия C++ см. в приложении к [Orwell, 1949].

Я начал разрабатывать C++ с тем, чтобы я и мои друзья перестали программировать на ассемблере, С или иных высокоуровневых языках того времени. Главной целью было сделать процесс создания качественных программ более простым и приятным. В первые годы не было никаких спецификаций и планов на бумаге: разработка, документирование и реализация выполнялись одновременно. Не было никакого «C++ проекта», равно как и «комитета по C++». Все это время C++ эволюционировал, помогая решать проблемы, возникающие у пользователей, а также под влиянием дискуссий с друзьями и коллегами.

Последовавший затем взрывной рост практического использования C++ привел к изменению ситуации. Где-то в 1987 году стало ясно, что формальная стандартизация C++ неизбежна, и мы стали готовить почву для этой деятельности [Stroustrup, 1994]. Результатом стали осознанные усилия по поддержанию контактов между производителями компиляторов и основными пользователями посредством обычной и электронной почты, а также на конференциях по C++ и в иных местах.

AT&T Bell Laboratories внесла значительный вклад в этот процесс, позволив мне делиться предварительными версиями спецификаций языка C++ с разработчиками компиляторов и пользователями. Поскольку многие из них работали на конкурентов AT&T, значение этого вклада не стоит недооценивать. Менее цивилизованная компания могла бы существенно попортить стандартизацию языка, просто ничего не делая. Сотни людей из десятков организаций прочли то, что потом стало общепризнанным руководством по языку и базовым документом для стандарта ANSI C++. Их имена перечислены в The Annotated C++ Reference Manual [Ellis, 1989]. В конце концов, в декабре 1989 года по инициативе Hewlett-Packard был создан комитет X3J16 при ANSI. В июне 1991 года эта инициатива ANSI (Американский Национальный Институт Стандартизации) стала частью международной деятельности под эгидой ISO (Международная Организация по Стандартизации). С 1990 года эти комитеты стали основным форумом для обсуждения эволюции и совершенствования языка C++. Я работал в этих комитетах в качестве главы рабочей группы по расширениям языка, где непосредственно отвечал за рассмотрение предложений по коренным изменениям и добавлению новых средств. Предварительный вариант

стандарта был выпущен для публичного рассмотрения в апреле 1995 года. Официальный стандарт ISO C++ (ISO/IEC 14882) был принят в 1998 году. Технический документ с исправлениями мельчайших ошибок и неточностей был опубликован в 2003 году [C++, 2003].

Эволюция C++ проходила рука об руку с некоторыми из классов, представленных в данной книге. Например, я разработал классы complex, vector и stack, наряду с механизмом перегрузки операций. Классы string и list были разработаны Джонатаном Шопиро (Jonathan Shopiro) и мною. Они стали первыми активно использовавшимися библиотечными классами. Современный класс string из стандартной библиотеки берет свое начало в этих ранних усилиях. Библиотека задач, рассмотренная в [Stroustrup, 1987] и в §12.7[11], была частью самой первой программы на «C с классами», написанной когда-либо. Она поддерживала стиль моделирования (simulation), присущий языку Simula. Библиотека задач неоднократно пересматривалась и изменялась, преимущественно Джонатаном Шопиро, и по сей день используется активно. Вариант потоковой библиотеки, рассмотренный в первом издании книги, был написан мною. Джерри Шварц (Jerry Schwarz) переработал его в стандартные классы потокового ввода/вывода (глава 21), используя технику манипуляторов Эндрю Кенига (Andrew Koenig) (§21.4.6) и некоторые другие идеи. Классы потокового ввода/ вывода подвергались в дальнейшем определенной переработке в процессе стандартизации, и большую часть работы выполнили Джерри Шварц, Натан Майерс (Nathan Myers) и Норихиро Кумагаи (Norihiro Kumagai). Разработка шаблонов испытала влияние шаблонных реализаций ***vector***, ***map***, ***list*** и ***sort***, выполненных Эндрю Кенигом, Александром Степановым и мною. В свою очередь, работа Степанова по обобщенному программированию на базе шаблонов привела к контейнерам и алгоритмам стандартной библиотеки (§16.3, глава 17, глава 18, §19.2). Предназначенные для численных расчетов классы стандартной библиотеки (например, ***valarray*** — см. главу 22) были разработаны, главным образом, Кентом Баджем (Kent Budge).

1.5. Применение C++

C++ используется сотнями тысяч программистов практически во всех предметных областях. Эта деятельность подкрепляется наличием десятков независимых реализаций, сотнями библиотек и книг, несколькими специализированными журналами, многими конференциями и бесчисленными консультантами. Широко доступны многочисленные учебные курсы по C++ самого разного уровня.

Ранние приложения на C++ явно тяготели к системным задачам. Несколько важных операционных систем целиком были написаны на C++ [Campbell, 1987], [Rozier, 1988], [Hamilton, 1993], [Berg, 1995], [Parrington, 1995], а еще большее количество имеет важные части, выполненные на C++. Я всегда рассматривал бескомпромиссную низкоуровневую эффективность важной для C++. Это позволяет писать на C++ драйверы и другие программные компоненты, непосредственно оперирующие аппаратурой компьютера в реальном масштабе времени. В таких программах предсказуемость поведения ничуть не менее важна, чем скорость выполнения. Часто и компактность результирующего кода имеет важное значение. C++ был разработан таким образом, что буквально каждое его средство может использоваться в задачах с суровыми требованиями к производительности и размеру [Stroustrup, 1994, §4.5].

Большинство приложений безусловно содержат участки кода, критичные с точки зрения производительности. Однако большая часть программного кода расположена не в них. Для большей части программного кода чрезвычайно важны простота сопровождения, расширения и тестирования. C++ нашел широчайшее применение в областях, где очень важна надежность, и где требования к программам часто меняются. Таковы торговля, банковские и страховые операции, телекоммуникации, а также военные приложения. Многие годы централизованное управление междугородними телефонными звонками в США осуществлялось программой, написанной на C++, и каждый звонок на номер с префиксом 800 обрабатывался этой программой [Kamath, 1993]. Такие приложения велики и эксплуатируются очень долго. Поэтому стабильность, масштабируемость и совместимость были непременными аспектами разработки C++. Программы, содержащие миллионы строк кода на C++, встречаются не так уж и редко.

Как и C, язык C++ не предполагал его интенсивного применения для решения вычислительных задач. И тем не менее, многие численные, научные и инженерные задачи программируются на C++. Основной причиной этого факта является необходимость комбинировать в программах вычислительную активность и графику, а также необходимость вычислений со сложными структурами данных, которые плохо вписываются в прокрустово ложе языка Fortran [Budge, 1992], [Barton, 1994]. Графика и GUI (Graphics User Interface — графический интерфейс пользователя) — вот области, где C++ используется весьма интенсивно. Любой, кто работал на Apple Macintosh или PC Windows, косвенно использовал C++, поскольку базовые пользовательские интерфейсы этих систем реализованы на C++. Наиболее популярные библиотеки для работы с графическим сервером X-System на платформе UNIX также написаны на C++. Таким образом, C++ широко используется в приложениях, для которых важен развитый пользовательский интерфейс.

Все перечисленное указывает на наиболее сильную сторону C++ — возможность эффективно обслуживать широкий диапазон прикладных областей. Не так уж трудно повстречаться с задачами, для которых одновременно нужны и сетевые взаимодействия (в локальных и глобальных сетях), и вычисления, и графика, и взаимодействие с пользователем, и доступ к базам данных. Традиционно эти области обслуживались раздельно и разными группами технических специалистов, каждая из которых применяла разные языки программирования. Язык же C++ одинаково хорошо применим ко всем этим областям. Более того, он разрешает сосуществовать фрагментам программы, написанным на разных языках.

C++ широко используется для обучения и исследований. Это вызывает удивление у многих людей, которые совершенно справедливо замечают, что это далеко не самый маленький и простой язык. И тем не менее, про C++ можно сказать, что он:

- достаточно ясен для обучения базовым концепциям,
- практичен, эффективен и гибок для создания особо критичных систем,
- доступен для организаций и коллективов, которым необходимы разнообразные среды разработки и выполнения,
- достаточно обширен и содержателен для изучения сложных концепций и методов,
- достаточно технологичен для переноса изученного в коммерческую среду.

В общем, C++ — это язык, вместе с которым можно расти непрерывно.

1.6. Языки С и С++

Язык С был выбран в качестве базы при разработке С++, потому что он:

1. Универсальный, лаконичный и относительно низкоуровневый.
2. Адекватен большей части системных задач.
3. Исполняется везде и на всем.
4. Хорошо подходит для программирования в среде UNIX.

Конечно, С имеет свои проблемы, но их имел бы и любой другой язык, специально разработанный с самого нуля, а проблемы С всем уже хорошо известны. Разработанный поверх С язык «С с классами» стал полезным (хотя и неуклюжим) инструментом практически в первые же месяцы после того, как вообще возникла идея добавить Simula-подобные классы в С. Однако по мере того, как росла популярность языка С++ и все более важными становились его средства, построенные поверх С, постоянно поднимался вопрос о целесообразности сохранения такой совместимости. Конечно, при устранении наследственности часть проблем отпадет сама собой (см., например, [Sethi, 1981]). Но все же мы на это не пошли по следующим причинам:

1. Миллионы строк кода на С могут эффективно инкорпорироваться в С++ программы, пока не требуется их значительной переделки.
2. Миллионы строк библиотечного кода и полезных утилит, написанных на С, могут использоваться из программ на С++, пока эти языки синтаксически близки и совместимы на уровне правил компоновки.
3. Сотни тысяч программистов знают С, так что для перехода на С++ им нужно изучить лишь новые конструкции С++, а не переучивать основы.
4. С и С++ будут долгие годы использоваться на одних и тех же системах одними и теми же людьми, так что разница между этими языками должна быть или минимальной, или очень большой, иначе не избежать путаницы и ошибок.

Определения в С++ были пересмотрены с тем, чтобы любая конструкция, применяемая и в С, и в С++, имела бы одинаковый смысл в обоих языках (с незначительными исключениями; см. §B.2)

Язык С также не стоял на месте и эволюционировал, частично под влиянием С++ [Rosler, 1984]. Стандарт ANSI С [С, 1999] содержит синтаксис объявления функций, позаимствованный у «С с классами». Заимствование шло в обоих направлениях. Например, тип ***void**** был впервые предложен для ANSI С, но реализован сначала в С++. Как было обещано в первом издании настоящей книги, определение С++ пересмотрено с целью устранения необоснованных несовместимостей, и теперь С++ более совместим С, чем вначале. В идеале, С++ должен быть близок к С настолько, насколько это возможно (но не более того) [Koenig, 1989]. Стопроцентная совместимость никогда не планировалась, ибо она подрывает типовую безопасность и гармонию между встроенными и пользовательскими типами.

Знание С не является обязательным условием для изучения С++. Программирование на С толкает на применение трюков, совершенно ненужных в свете возможностей С++. В частности, явное приведение типов в С++ используется реже, чем

в C (§1.6.1). В то же время, *хорошие* программы на C, по сути, весьма близки к программам на C++. Таковы, например, все программы из книги Кернигана и Ритчи *The C Programming Language (2nd Edition)* [Kernigan, 1988]. В принципе, знание любого языка со статической системой типов помогает изучению C++.

1.6.1. Информация для C-программистов

Чем лучше вы знакомы с C, тем труднее вам будет удержаться от соблазна писать на C++ в стиле C, теряя тем самым многие преимущества C++. Пожалуйста, ознакомьтесь с приложением B, в котором перечисляются все различия между C и C++. Вот некоторые вещи, с которыми C++ справляется лучше, чем C:

1. Макросы менее необходимы в C++. Используйте ***const*** (§5.4) или ***enum*** (§4.8) для объявления констант, ***inline*** (§7.1.1) — чтобы избежать накладных расходов на вызов функции, ***шаблоны*** (глава 13) — для определения семейств функций и типов, и ***namespace*** (§8.2) для преодоления коллизии имен.
2. Объявляйте переменные только тогда, когда они потребуются и тотчас же инициализируйте их. В языке C++ объявление допустимо всюду, где может стоять оператор (§6.3.1), в инициализирующих выражениях в операторе цикла ***for*** (§6.3.3) и в условиях (§6.3.2.1).
3. Не применяйте ***malloc***(). Операция ***new*** (§6..2.6) работает лучше. Вместо ***realloc***() попробуйте тип ***vector*** (§3.8, §16.3).
4. Старайтесь избегать ***void****, арифметики указателей, объединений, явных приведений типа кроме как глубоко внутри реализаций функции или класса. В большинстве случаев явное приведение типов свидетельствует о дефектах проектирования. Если преобразование все же нужно, применяйте одну из новых операций приведения (§6.2.7), которые точнее отражают ваши намерения.
5. Минимизируйте применение массивов и строк в C-стиле — стандартные библиотечные типы ***string*** (§3.5) и ***vector*** (§3.7.1) заметно упрощают программирование. В целом, не пытайтесь программировать самостоятельно то, что уже запрограммировано в стандартной библиотеке.

Чтобы обеспечить компоновку функции C++ по правилам языка C, применяйте в ее объявлении соответствующий модификатор (§9.2.4).

И что самое важное, старайтесь мыслить о программе как о наборе взаимосвязанных концепций, представимых классами и объектами, а не о наборе структур данных, чьи биты перерабатываются функциями.

1.6.2. Информация для C++-программистов

Многие программисты используют C++ уже целое десятилетие. Большинство из них работают в одной и той же среде и привыкли к ограничениям, свойственным ранним компиляторам и библиотекам. Нельзя сказать, что опытные программисты не замечали появления новых средств в C++ — скорее, они не отслеживали их взаимоотношений, вследствие которых и стали возможными новые фундаментальные технологии. То, что ранее было неизвестно или казалось непрактичным, теперь может оказаться наилучшим подходом. Выяснить это можно лишь пересмотрев основы.

Читайте главы по порядку. Если содержание главы вам известно, то ее беглый просмотр займет лишь несколько минут. а если нет, то тогда вы наверняка столкнетесь с неожиданностями. Я сам кое-что узнал, пока писал эту книгу, и полагаю, что вряд ли существует такой программист, который знает абсолютно все изложенные здесь средства и технологии. Более того, чтобы правильно пользоваться языком, нужно видеть перспективу, которая и вносит порядок в набор средств и технологий. Данная книга с ее организацией и примерами нацелена на то, чтобы дать вам такую перспективу.

1.7. Размышления о программировании на C++

В идеале вы разрабатываете программу в три этапа. Сначала вы обдумываете исходную постановку задачи (анализ), затем вы формулируете ключевые концепции, связанные с решением задачи (проектирование), и только после этого воплощаете решение в виде программы (программирование). К сожалению, часто бывает так, что и проблема, и методы ее решения становятся предельно ясными лишь в процессе написания и отладки работающей программы. Вот здесь-то и сказывается выбор языка.

В большинстве приложений встречаются понятия, отразить которые в терминах фундаментальных типов или в виде функции без ассоциирированных с нею данных весьма затруднительно. Столкнувшись с таким понятием, вводите класс, который и будет отражать это понятие в программе. Класс языка C++ — это тип. Тип специфицирует поведение объектов этого типа: как они создаются, как ими можно манипулировать и как объекты уничтожаются. Класс также определяет представление объектов, хотя на ранней стадии проектирования этот вопрос не является основным. Залогом хорошего проектирования служит четкое соответствие: «одно понятие — один класс». Для этого часто приходится сосредоточиться на вопросах типа: «Как создаются объекты рассматриваемого класса? Могут ли объекты класса копироваться и/или уничтожаться? Какие операции могут выполняться над этими объектами?» Если четких ответов на эти вопросы нет, то скорее всего нет и ясного понимания исходного понятия. Лучше вернуться к осмыслению исходной задачи и переформулировке основных концепций ее решения, а не кидаться тут же программировать с неясными последствиями.

Традиционно, легче всего работать с объектами, имеющими ярко выраженную математическую природу: числами всех типов, множествами, геометрическими фигурами и т.п. Текстовый ввод/вывод, строки, базовые контейнеры, фундаментальные алгоритмы обработки контейнеров и ряд математических классов являются частью стандартной библиотеки C++ (глава 3, §16.1.2). Кроме того, существует огромное количество библиотек, реализующих как самые общие, так и узкоспецифические для конкретных предметных областей понятия.

Понятия не существуют в вакууме; чаще всего они взаимодействуют друг с другом. Часто выявить и точно определить характер взаимодействия между понятиями бывает труднее, чем сформулировать набор понятий самих по себе. Лучше избегать «каши» из классов (понятий), взаимодействующих по принципу «каждый с каждым». Рассмотрим два класса, A и B. Отношения «A вызывает функции из B», «A создает объекты типа B» и «член класса A имеет тип B» редко создают проблемы.

В то же время, отношений «А использует данные из В» в общем случае следует избегать.

Одним из мощнейших интеллектуальных приемов преодоления сложностей, связанных с иерархическими зависимостями типов, является их упорядочение в древовидную структуру с самым общим понятием в корне дерева. В языке С++ такое упорядочение осуществляется с помощью механизма наследования классов. Часто программу можно организовать в виде набора деревьев или ациклических направленных графов классов. При этом программист определяет набор базовых классов, у каждого из которых имеются свои производные классы. Виртуальные функции (§2.5.5, §12.2.6) широко применяются для определения операций с наиболее общими понятиями (базовыми классами). Если необходимо, то эти операции могут уточняться для более специализированных понятий (производных классов).

Но иногда даже ациклических направленных графов не хватает для организаций понятий (концепций) конкретной программы; например, когда часть понятий представляются взаимозависимыми по своей природе. Тогда нужно локализовать циклические зависимости, чтобы они не влияли на структуру программы в целом. Если же не удается ни избавиться от циклических зависимостей, ни локализовать их, то вы, похоже, находитесь в тупике, выйти из которого вам никакой язык не поможет. В самом общем случае, если не удается сформулировать относительно простые связи между базовыми понятиями задачи, то программа, скорее всего, будет неуправляемой.

Одним из лучших способов распутывания взаимозависимостей является четкое разделение интерфейса и реализации. Абстрактные классы языка С++ (§2.5.4, §12.3) служат решению этой задачи.

Другая форма общности может быть выражена с помощью шаблонов (§2.7, глава 13). Классовый шаблон определяет целое семейство классов. Например, шаблон списков определяет «список элементов типа Т», где Т может быть любым типом. Таким образом, шаблон — это механизм порождения типа на основе другого типа, передаваемого шаблону в качестве параметра. Наиболее распространенными шаблонами являются контейнеры (такие как списки, вектора и ассоциативные массивы), а также фундаментальные алгоритмы для работы с этими контейнерами. Типовую параметризацию классов и связанных с ними функций всегда лучше выражать с помощью шаблонов, а не через механизм наследования классов.

Помните, что большое количество программ можно просто и ясно написать, используя лишь примитивные типы, структуры данных, традиционные функции и небольшое количество библиотечных классов. Весь мощнейший аппарат определения новых типов следует применять лишь там, где в этом есть серьезная необходимость.

Вопрос «Как писать хорошие программы на С++?» практически полностью аналогичен вопросу « Как писать хорошую английскую прозу?». Здесь есть два совета: «Знай то, что хочешь сказать» и «Практикуйся. Подражай успешным образцам». Оба совета походят и для С++, и для английского языка, но обоим из них так трудно следовать.

1.8. Советы

Ниже дан набор «правил», которые могут пригодиться при изучении C++. По мере продвижения в этом процессе вы можете переработать их в нечто, более пригодное для ваших задач и вашего индивидуального стиля программирования. Перечисленные правила являются упрощенными и не отражают множества деталей. Не воспринимайте их слишком буквально. Для написания хороших программ требуются ум, вкус и терпение. Вряд ли эти правила помогут вам с самого начала. Экспериментируйте!

1. В процессе программирования вы реализуете конкретное представлений концепций решения некоторой проблемы. Постарайтесь, чтобы структура программы отражала эти концепции как можно более непосредственно:
   - [a] Если вы мыслите об «этом» как о некой общей сущности, сделайте «это» классом.
   - [b] Если вы мыслите об «этом» как о конкретном экземпляре сущности, сделайте «это» объектом класса.
   - [c] Если два класса имеют общий интерфейс, оформите его в виде абстрактного класса.
   - [d] Если реализации двух классов имеют что-то общее, вынесите это общее в базовый класс.
   - [e] Если класс является контейнером объектов, сделайте его шаблоном.
   - [f] Если функция реализует алгоритм работы с контейнером, сделайте ее функциональным шаблоном, реализующим алгоритм для работы с семейством контейнеров.
   - [g] Если классы, шаблоны и т.д. логически связаны, поместите их все в единое пространство имен.
2. Определяя класс (если только это не математический класс вроде матриц или комплексных чисел, или низкоуровневый тип вроде связного списка):
   - [a] Не используйте глобальные данные (применяйте классовые поля данных).
   - [b] Не используйте глобальные функции.
   - [c] Не делайте поля данных открытыми.
   - [d] Не используйте дружественные функции, разве что во избежание [a] и [c].
   - [e] Не используйте в классах «поля типа»; применяйте виртуальные функции.
   - [f] Не используйте встраиваемые функции, разве что для значительной оптимизации.

Более конкретные или подробные правила вы найдете в разделе советов в конце каждой главы. Помните, что это лишь советы, а не непреложные законы. Советами стоит пользоваться только там, где это разумно. Они не заменяют ум, опыт, здравый смысл и хороший вкус.

Я считаю советы вроде «не делайте этого никогда» бесполезными. Поэтому большинство советов формулируются в виде предложений, что желательно делать, а негативные утверждения не формулируются в виде абсолютных запретов. Я не знаю ни одного важного элемента языка C++, примеры полезного применения которого на практике не встречаются. Раздел «Советы» не содержит пояснений. Вместо этого, каждый совет снабжается ссылкой на соответствующий раздел книги. В случае негативных советов, указанный раздел содержит позитивную альтернативу.

1.8.1. Литература

Несколько прямых ссылок есть в тексте, а здесь мы приводим полный список книг и статей, упомянутых прямо или косвенно.

[Barton, 1994] John J. Barton, and Lee R. Nackman: *Scientific and Engineering C++*, Addison-Wesley. Reading, Mass. 1994. ISBN 1-201-53393-6.

[Berg, 1995] William Berg, Marshall Cline, and Mike Girou: *Lessons Learned from the OS/400 00 Project.* CACM. Vol. 38 No. 10. October 1995.

[Booch, 1994] Grady Booch: *Object-Oriented Analysis and Design,* Benjamin/Cummings. Menlo Park, Calif. 1994. ISBN 0-8053-5340-2.

[Budge, 1992] Kent Budge, J. S. Perry, and A. C. Robinson: *High-Performance Scientific Computation using C++.* Proc. USENIX C++ Conference. Portland, Oregon. August 1992.

[C, 1990] X3 Secretariat: *Standard — The C Language.* X3J11/90-013. ISO Standard ISO/IEC 9899. Computer and Business Equipment Manufacturers Association. Washington, DC, USA.

[C++, 1998] X3 Secretariat: *International Standard — The C++ Language* X3J16-14882. Information Technology Council (NSITC)] Washington, DC, USA.

[C++, 2003] X3 Secretariat: *International Standard — The C++ Language* (including the 2003 Technical Corrigendum). 14882:2003(E). Information Technology Council (NSITC). Washington, DC, USA.

[Campbell, 1987] Roy Campbell, et al.: *The Design of a Multiprocessor Operating System.* Proc. USENIX C++ Conference. Santa Fe, New Mexico November 1987.

[Coplien, 1995] James O. Coplien and Douglas C. Schmidt (editors): *Pattern Languages of Program Design.* Addison-Wesley. Reading. Mass, 1995. ISBN 1-201-60734-4

[Dahl, 1970] O-J. Dahl, B. Myrhaug, and K. Nygaard: *SIMULA Common Base Language.* Norwegian Computing Center S-22. Oslo, Norway. 1970.

[Dahl, 1972] O-J. Dahl and C A. R. Hoare: *Hierarchical Program Construction in Structured Programming.* Academic Press, New York. 1972.

[Ellis, 1989] Margaret A. Ellis and Bjarne Stroustrup: *The Annotated C++ Reference Manual.* Addison-Wesley. Reading, Mass. 1990. ISBN 0-201-51459-1.

[Gamma, 1995] Erich Gamma, et al.: *Design Patterns.* Addison-Wesley. Reading, Mass. 1994. ISBN 0-201-63361-2.

[Goldberg, 1983] A. Goldberg and D. Robson: *SMALLTALK-80 — The Language and Its Implementation.* Addison-Wesley. Reading, Mass. 1983.

[Griswold, 1970] R. E. Griswold, et al.: *The Snobol4 Programming Language.* Prentice-Hall. Englewood Cliffs, New Jersey. 1970.

[Hamilton, 1993] G. Hamilton and P. Kougiouris: *The Spring Nucleus: A Microkernel for Object.* Proc. 1993 Summer USENIX Conference. USENIX.

[Griswold, 1983] R. E. Griswold and M. T. Griswold: *The ICON Programming Language.* Prentice-Hall. Engewood Cliffs, New Jersey. 1983.

[Henricson, 1997] Mats Henricson and Erik Nyquist: *Industrial Strength C++: Rules and Recommendations.* Prentice-Hall. Engewood Cliffs, New Jersey. 1997. ISBN 0-13-120965-5.

[Ichbian, 1979] Jean D. Ichbiah, et al.: *Rationale for the Design of the ADA Programming Language.* SIGPLAN Notices. Vol. 14 No. 6. June 1979.

[Kamath, 1993] Yogeesh H. Kamath, Ruth E. Smilan, and Jean G. Smith: *Reaping Benefits with Object-Oriented Technology.* AT&T Technocal Journal. Vol. 72 No. 5. September/October 1993.

[Kernighan, 1978] Brian W. Kernighan and Dennis M. Ritchie: *The C Programming Language.* Prentice-Hall. Englewood Cliffs, New Jersey. 1978.

[Kernighan, 1988] Brian W. Kernighan and Dennis M. Ritchie: *The C Programming Language (SecondEdition).* Prentice-Hall. Englewood Cliffs, New Jersey. 1978. ISBN 0-13-110362-8.

[Koenig, 1989] Andrew Koenig and Bjarne Stroustrup: *C++: As close to C as possible — but no closer.* The C++ Report. Vol. 1 No. 7. July 1989.

[Koenig, 1997] Andrew Koenig and Barbara Moo: *Ruminations on C++.* Addison Wesley Longman. Reading, Mass. 1997. ISBN 1-201-42339-1.

[Knuth, 1968] Donald Knuth: *The Art of Computer Programming.* Addison-Wesley. Reading, Mass.

[Liskov, 1979] Barbara Liskov et al.: *Clu Reference Manual.* MIT/LCS/TR-225. MIT Cambridge. Mass. 1979.

[Martin, 1995] Robert C Martin: *Designing Object-Oriented C++ Applications Using the Booch Method.* Prentice-Hali. Englewood Cliffs, New Jersey. 1995. ISBN 0-13-203837-4.

[Orwell, 1949] George Orwell: *1984.* Secker and Warburg. London. 1949

[Parrington, 1995] Graham Parrington et al.: *The Design and Implementation of Arjuna.* Computer Systems. Vol. 8 No. 3. Summer 1995.

[Richards, 1980] Martin Richards and Colin Whitby-Strevens: *BCPL — The Language and Its Compiler.* Cambridge University Press, Cambridge. England. 1980. ISBN 0-521-21965-5.

[Rosier, 1984] L. Rosier: *The Evolution of C — Past and Future.* AT&T Bell Laboratories Technical Journal. Vol. 63 No. 8. Part 2. October 1984.

[Rozier, 1988] M. Rozier, et al.: *CHORUS Distributed Operating Systems.* Computing Systems. Vol. 1 No. 4. Fall 1988.

[Sethi, 1981] Ravi Sethi: *Uniform Syntax for Type Expressions and Declarations.* Software Practice & Experience. Vol. 11. 1981.

[Stepanov, 1994] Alexander Stepanov and Meng Lee: *The Standard Template Library.* HP Labs Technical Report HPL-94-34 (R. 1). August, 1994.

[Stroustrup, 1986] Bjarne Stroustrup: *The C++ Programming Language.* Addison-Wesley. Reading, Mass. 1986. ISBN 0-201-12078-X.

[Stroustrup, 1987] Bjarne Stroustrup and Jonathan Shopiro: *A Set of C Classes for Co-Routine Style Programming.* Proc. USENIX C++ Conference. Santa Fe, New Mexico. November 1987.

[Stroustrup, 1991] Bjarne Stroustrup: *The C++ Programming Language (Second Edition).* Addison-Wesley. Reading, Mass. 1991. ISBN 0-201-53992-6.

[Stroustrup, 1994] Bjarne Stroustrup: *The Design and Evolution of C++.* Addison-Wesley. Reading, Mass. 1994. ISBN 0-2-1-54330-3.

[Tarjan, 1983] Robert E. Tarjan: *Data Structures and Network Algorithms.* Society for Industrial and Applied Mathematics. Philadelphia, Penn. 1983. ISBN 0-898-71187-8.

[Unicode, 1996] The Unicode Consortium: *The Unicode Standard, Version 2.0.* Addison-Wesley Developers Press. Reading, Mass, 1996. ISBN 0-201-48345-9.

[UNIX, 1985] UNIX Time-Sharing System: *Programmer's Manual. Research Version, Tenth Edition.* AT&T Bell Laboratories, Murray Hill, New Jersey. February 1985.

[Wilson, 1996] Gregory V. Wilson and Paul Lu (editors): *Parallel Programming Using C++* . The MIT Press. Cambridge. Mass. 1996. ISBN 0-262-73118-5.

[Wikstrom, 1987] Ake Wikstrom: *Functional Programming Using ML.* Prentice-Hall Englewood Cliffs, New Jersey. 1987.

[Woodward, 1974] P. M. Woodward and S. G. Bond: *Algol 68-R Users Guide.* Her Majesty's Stationery Office. London. England. 1974.

Русские переводы:

[Booch, 1994] Г. Буч. *Объектно-ориентированный анализ и проектирование с примерами приложений на С++, 2-е издание.* СПб. «Невский Диалект». 1998.

[Ellis, 1989] Б. Страуструп, М. А. Эллис. *Справочное руководство по языку С++ с комментариями: проект стандарта ANSI.* М. «Мир». 1992.

[Kernighan, 1988] Б. Керниган, Д. Ричи. *Язык программирования Си. 3-е издание.* СПб. «Невский Диалект». 2001.

[Knuth, 1968] Д. Кнут. *Искусство программирования для ЭВМ. т. 1, 2, 3.* М. «Мир». 1976.

[Orwell, 1949] Дж. Оруэлл. *1984* (см. в сборнике: Дж. Оруэлл. *Проза отчаяния и надежды.* СПб. «Лениздат». 1990).

[Stroustrup, 1986] Б. Страуструп. *Язык программирования Си++.* М. «Радио и связь». 1991.

Ссылки на литературу, посвященную проектированию и другим проблемам разработки крупных программных проектов, можно найти в конце главы 23.

2

Обзор языка C++

В первую очередь, разделаемся с защитниками языка.
— Генрих VI, Часть II

https://t.me/it_boooks

Что такое C++? — парадигмы программирования — процедурное программирование — модульность — раздельная компиляция — обработка исключений — абстракция данных — типы, определяемые пользователем — конкретные типы — абстрактные типы — виртуальные функции — объектно-ориентированное программирование — обобщенное программирование — контейнеры — алгоритмы — язык и программирование — советы.

2.1. Что такое язык C++?

Язык C++ — это универсальный язык программирования с акцентом на системном программировании, и который:

- лучше, чем язык C,
- поддерживает абстракции данных,
- поддерживает объектно-ориентированное программирование,
- поддерживает обобщенное программирование.

В настоящей главе поясняется, что все это значит, но без ознакомления с мельчайшими деталями языковых определений. Дается общий обзор языка C++ и ключевых идей его применения, не отягощенных подробными сведениями, необходимыми для начала реального практического программирования на C++.

Если восприятие отдельных мест настоящей главы покажется вам затруднительным, не обращайте внимание и просто идите дальше. В последующих главах все разъясняется в деталях.

Неплохо, однако, потом вернуться к пропущенному еще раз. Детальное знание всех языковых свойств и конструкций, даже их исчерпывающее знание, не заменяет собой и не отменяет необходимость общего понимания роли и места этого языка

программирования, а также знакомства с фундаментальными идеями и приемами его использования.

2.2. Парадигмы программирования

Объектно-ориентированное программирование есть техника программирования, можно сказать, парадигма, для написания «хороших» программ для некоторого класса задач. Термин же «объектно-ориентированный язык программирования» должен в первую очередь означать такой язык, который хорошо поддерживает объектно-ориентированный стиль программирования.

Здесь следует четко различать две ситуации. Мы говорим, что язык *поддерживает* некоторый стиль программирования, если он предоставляет специальные средства, позволяющие этот стиль легко (удобно, надежно, эффективно) реализовывать на практике. Если же язык требует исключительных усилий и огромного опыта для реализации некоторой техники программирования, то мы говорим, что он лишь *допускает* возможность ее применения. Например, на языке Fortran 77 можно писать структурные программы, а на языке С можно писать программы в объектно-ориентированном стиле, но все это требует неадекватно больших усилий, так как эти языки напрямую не поддерживают указанные стили программирования.

Поддержка парадигмы программирования осуществляется не только через специально для этого предназначенные языковые конструкции, но и более тонко — через механизмы автоматического выявления непреднамеренного отклонения от выбранной парадигмы как на этапе компиляции, так и/или на этапе выполнения программы. Строгая проверка типов является очевидным примером такого механизма; выявление неоднозначностей (двусмысленностей — ambiguities) и контроль исполнения программы дополняют языковые средства поддержки парадигмы программирования. Внеязыковые средства, такие как библиотеки и среды программирования, еще более расширяют реальную поддержку парадигмы.

В то же время, неправильно говорить, что некоторый язык программирования лучше другого просто потому, что он содержит некоторые конструкции, отсутствующие в другом языке. И тому есть немало реальных примеров. На самом деле, важно не то, какими средствами обладает язык, а то, что средств этих на деле достаточно для реальной поддержки стиля программирования, оптимального для конкретной предметной области:

1. Все эти средства должны естественно и элегантно интегрироваться в язык.
2. Должна иметься возможность комбинировать средства для достижения целей, в противном случае требующих применения дополнительных специализированных средств.
3. Неестественных «средств специального назначения» в языке должно быть как можно меньше.
4. Реализация средств поддержки парадигм не должна обременять не нуждающиеся в них программы.
5. От программиста нельзя требовать знаний тех аспектов языка, которых он явным образом не использует в своих программах.

Первый из перечисленных принципов апеллирует к эстетике и логике. Два последующих выражают концепцию минимализма. А два последних принципа гласят: «нет нужды — нет проблемы».

Язык C++ создавался с учетом перечисленных принципов как расширение языка C для добавления поддержки абстракции данных, объектно-ориентированного и обобщенного стилей программирования. Он *не навязывает* программисту конкретный стиль программирования.

Стили программирования и поддерживающие их механизмы языка C++ рассматриваются в последующих разделах. Изложение материала стартует с рассмотрения процедурного стиля программирования, а заканчивается объектно-ориентированным программированием классовых иерархий и обобщенным программированием, использующим шаблоны языка C++. Показано, что каждая из парадигм строится поверх ее предшественницы, каждая вносит что-то новое в совокупный рабочий инструментарий C++-программиста, и что каждая парадигма основывается на проверенных временем и практикой подходах к проектированию и разработке программ.

Изложение материала в последующих разделах настоящей главы не является исчерпывающим. Акцент делается на вопросах проектирования программ и на их структуре, а вовсе не на тонкостях и деталях самого языка. На данном этапе важнее понять, что именно можно сделать с помощью языка C++, а не то, как этого достичь практически.

2.3. Процедурное программирование

Исходная парадигма процедурного программирования звучит так:

Решите, какие процедуры нужны;
используйте наилучшие алгоритмы.

Здесь акцент делается на процессе обработки данных, в первую очередь на алгоритме необходимых вычислений. Языки поддерживают эту парадигму, предоставляя механизмы передачи аргументов функциям и получения их возврата (то есть вычисленного значения функции). Литература по процедурному программированию наполнена обсуждением способов передачи аргументов и различением их типов, классификацией видов функций (процедур, подпрограмм и макросов) и т.п.

Функция извлечения квадратного корня является типичным примером «хорошего стиля» для процедурной парадигмы. Получив аргумент в виде числа с плавающей точкой двойной точности, она вырабатывает искомый результат. Для этого она выполняет хорошо известные математические вычисления:

```
double sqrt(double arg)
{
  // код для вычисления квадратного корня
}
void f()
```

```
{
  double root2 = sqrt(2) ;
  // ...
}
```

Фигурные скобки, { }, в языке C++ предназначены для группировки. Здесь они указывают начало и конец тела функции. Двойная наклонная черта, //, означает начало комментария, который занимает всю оставшуюся часть строки. Ключевое слово void указывает здесь на отсутствие у функции возвращаемого значения (возврата).

С точки зрения организации программ, функции используются для наведения порядка в хаосе алгоритмов. Сами алгоритмы записываются с помощью функциональных вызовов и других средств языка. В следующих подразделах бегло рассматриваются простейшие средства языка C++, предназначенные для организации вычислений.

2.3.1. Переменные и простейшая арифметика

Каждое имя в программе и каждое выражение имеют определенный тип, задающий набор допустимых для них операций. К примеру, следующее объявление

```
int inch;
```

устанавливает, что ***inch*** относится к типу ***int***, то есть ***inch*** является целочисленной переменной.

Объявление является *оператором* (*statement*), вводящим имя в программу. Оно связывает имя с типом. *Тип* определяет, что можно (допустимо) делать с именем или выражением.

Язык C++ имеет ряд *встроенных типов*, имеющих непосредственное отношение к устройству электронных узлов (процессор и оперативная память) компьютера. Например:

```
bool      // логический, возможные значения - true и false
char      // символьный, например, 'a' , 'z', и '9'
int       // целый, например, 1, 42, и 1216
double    // вещественные числа с двойной точностью, например, 3.14 и 299793.0
```

Для заданного компьютера переменная типа ***char*** имеет естественный размер, используемый для хранения символа (как правило, один байт), а переменная типа ***int*** — естественный размер, используемый для вычислений с целыми числами (обычно называемый *словом*).

С любой комбинацией этих типов можно выполнять следующие арифметические операции:

```
+       // плюс, унарный и бинарный
-       // минус, унарный и бинарный
*       // умножение
/       // деление
%       // остаток от деления (второй операнд не может иметь тип
        // с плавающей запятой)
```

А также операции сравнения:

```
==     // равно
!=     // не равно
<      // меньше чем
>      // больше чем
<=     // меньше или равно
>=     // больше или равно
```

Для арифметических операций и операций присваивания C++ выполняет имеющие очевидный смысл преобразования типов, так что их можно использовать смешанным образом:

```
void some_function ()   // функция, не возвращающая значение
{
  double d = 2.2;       // инициализировать число с плавающей запятой
  int i = 7;            // инициализировать целое число
  d = d+i;              // присвоить сумму переменной d
  i = d*i;              // присвоить произведение переменной i
}
```

Как и в языке C, знак = используется для операции присваивания, а знак == используется для операции сравнения на равенство.

2.3.2. Операторы ветвления и циклы

Язык C++ обеспечивает общепринятый набор операторов ветвления и цикла. Для примера рассмотрим функцию, выводящую приглашение к вводу и возвращающую булевское (логическое) значение, зависящее от ввода пользователя:

```
bool accept ()
{
  cout<< "Do you want to proceed (y or n) ?\n";  // вывести вопрос

  char answer = 0;
  cin >> answer;                                  // считать ответ
  if(answer ==  'y') return true;
  return false;
}
```

Операция << (*«вставить в поток»*) используется как операция вывода, а **cout** представляет собой стандартный поток вывода. *Операция* >> (*«извлечь из потока»*) используется как операция ввода, а **cin** представляет собой стандартный поток ввода. Правый операнд операции >> принимает введенное значение, причем само это значение зависит от типа операнда. Символ **\n** в конце выводимой строки означает переход на новую строку.

Приведенный пример можно слегка улучшить, если явно отреагировать на ответ 'n':

```
bool accept2 ()
{
  cout << "Do you want to proced (y or n) ?\n";  // вывести вопрос

  char answer = 0;
  cin>>answer;                                   // считать ответ
```

```
  switch (answer)
  {
    case 'y' :
      return true;
    case 'n' :
      return false;
    default:
      cout<< "I'll take that for a no.\n";
      return false;
  }
}
```

Оператор **switch** сравнивает заданное значение с набором констант. Константы в разных ***case***-ветвях должны быть разными, и если проверяемое значение не совпадает ни с одной из них, то выбирается ***default***-ветвь. Программист, в общем случае, не обязан реализовывать ***default***-ветвь.

Редкая программа совсем не использует циклов. В нашем примере мы могли бы дать пользователю несколько попыток ввода:

```
bool accept3 ()
{
  int tries = 1;
  while (tries < 4)
  {
    cout << "Do you want to proceed (y or n)?\n";  // вывести вопрос
    char answer = 0;
    cin >> answer;                                  // считать ответ

    switch (answer)
    {
      case 'y' :
        return true;
      case 'n' :
        return false;
      default:
        cout << "Sorry, I don't understand that.\n";
        tries = tries + 1;
    }
  }
  cout << "I'll take that for a no.\n";
  return false;
}
```

Оператор цикла **while** исполняется до тех пор, пока его условие не станет равным ***false*** («*ложь*»).

2.3.3. Указатели и массивы

Массив можно объявить следующим образом:

```
char v[10];    // массив из 10 символов
```

Указатель объявляется так, как показано ниже:

```
char* p;    // указатель на символ
```

В этих объявлениях [] означает «*массив*», а * означает «*указатель*». Массивы индексируются с нуля, то есть ***v*** содержит следующие десять элементов: ***v***[***0***] ... ***v***[***9***]. Указатель может содержать адрес объекта соответствующего типа:

```
p = &v[3]; // p указывает на четвертый элемент массива v
```

Унарная операция & означает «взятие адреса».

Рассмотрим копирование десяти элементов массива в другой массив:

```
void another_function()
{
 int v1[10];
 int v2[10];
 // ...
 for(int i=0; i<10; ++i) v1[i]=v2[i];
}
```

Оператор цикла for задает следующую последовательность действий: «присвоить ***i*** значение ***0***; пока ***i*** меньше ***10***, копировать ***i***-ый элемент и инкрементировать ***i***». *Операция инкремента* ++ увеличивает значение целой переменной на единицу.

2.4. Модульное программирование

С течением времени акцент в разработке программ сместился от разработки процедур в сторону организации данных. Помимо прочего, этот сдвиг вызван увеличением размера программ. Набор связанных процедур и обрабатываемых ими данных часто называют *модулем* (*module*). Парадигма программирования теперь звучит так:

Решите, какие модули нужны;
сегментируйте программы так, чтобы данные были скрыты в модулях.

Эта парадигма известна также как *принцип сокрытия данных*. В задачах, где группировка процедур и данных не нужна, процедурный стиль программирования по-прежнему актуален. К тому же, принципы построения «хороших процедур» теперь применяются по отношению к отдельным процедурам модуля. Типовым примером модуля может служить определение стека. Вот соответствующий список задач, требующих решения:

1. Предоставить пользовательский интерфейс к стеку (например, функции ***push***() и ***pop***()).
2. Гарантировать доступ к внутренней структуре стека (реализуемой, например, с помощью массива элементов) исключительно через интерфейс пользователя.
3. Гарантировать автоматическую инициализацию стека до первого обращения пользователя.

Язык C++ предоставляет механизм группировки связанных данных, функций и т.д. в *пространства имен* (*namespaces*). Например, пользовательский интерфейс модуля ***Stack*** может быть объявлен и использован следующим образом:

```
namespace Stack   // интерфейс
{
  void push (char) ;
  char pop () ;
}

void f()
{
  Stack::push ('c') ;
  if(Stack::pop () != 'c') error ("impossible") ;
}
```

Квалификатор ***Stack::*** означает, что речь идет о функциях ***push*** () и ***pop*** (), определенных в пространстве имен ***Stack***. Иное использование этих имен не конфликтует с указанным и не вносит никакой путаницы в программу.

Полное определение стека можно разместить в отдельно компилируемой части программы:

```
namespace Stack   // реализация
{
  const int max_size= 200;
  chart v[max_size] ;
  int top = 0;

  void push (char c) { /* проверить на переполнение и push c */ }
  char pop () { /* проверить на пустоту стека и pop */ }
}
```

Ключевым в этом определении является тот факт, что пользовательский код изолируется от внутреннего представления модуля ***Stack*** посредством кода, реализующего функции ***Stack::push*** () и ***Stack::pop*** (). Пользователю нет необходимости знать, что ***Stack*** основан на массиве, и в результате имеется возможность изменять его внутреннее представление без последствий для пользовательского кода. Символы /* маркируют начало в общем случае многострочного комментария, заканчивающегося символами */.

Так как данные есть лишь часть того, что бывает желательным «скрыть», то концепция сокрытия данных очевидным образом расширяется до *концепции сокрытия информации*: имена функций, типы и т.д. локально определяются внутри модулей. Для этого язык C++ разрешает поместить в пространство имен любое объявление (§8.2).

Модуль ***Stack*** является лишь одной из возможных реализаций стека. Иные примеры стека в последующих главах будут иллюстрировать другие стили программирования.

2.4.1. Раздельная компиляция

Язык C++ поддерживает *принцип раздельной компиляции* (*separate compilation*), взятый из языка C. Это позволяет организовать программу в виде набора относительно независимых фрагментов.

В типовом случае, мы размещаем части модуля, ответственные за интерфейс взаимодействия с пользователем, в отдельный файл с «говорящим» именем. Так, код

```
namespace Stack  // интерфейс
{
  void push (char) ;
  char pop () ;
}
```

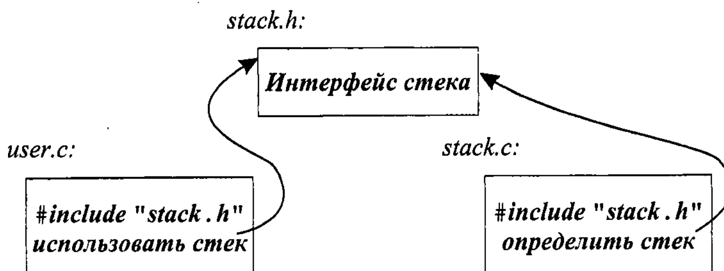
будет помещен в файл ***stack.h***, называемый *заголовочным файлом* (*header file*). Пользователи могут *включить* его в свой код следующим образом:

```
#include "stack.h" // включить интерфейс

void f()
{
  Stack::push ('c') ;
  if(Stack::pop () != 'c') error ("impossible") ;
}
```

Чтобы компилятор мог гарантировать согласованность кода, в файл с реализацией стека также нужно включить интерфейсный заголовочный файл:

```
#include "stack.h"  // включить интерфейс

namesspace Stack      // представление
{
  const int max_size= 200;
  char v[max_size] ;
  int top = 0;
}

void Stack::push (char c) { /* проверить на переполнение и push c */ }
char Stack::pop () { /* проверить на пустоту стека и pop */ }
```

Пользовательский код может располагаться в третьем файле, скажем, ***user.c***. Код файлов ***user.c*** и ***stack.c*** разделяет информацию об интерфейсе стека, расположенную в файле ***stack.h***. Во всем остальном файлы ***user.c*** и ***stack.c*** независимые и могут компилироваться раздельно. Графически, рассмотренные фрагменты программы можно представить следующим образом:



Раздельная компиляция является чрезвычайно полезным инструментом в реальном программировании. Это не узкая материя, с которой сталкиваются лишь программы, реализующие в виде модулей концепции стека и т.п. И хотя раздельная компиляция и не относится, строго говоря, к базовым элементам языка, она наилучшим образом позволяет извлечь максимальную пользу из его конкретных реализаций. Так или иначе, это вопрос большого практического значения. Целесообразно стремиться к достижению максимальной степени модульности программы, логической реализации этой модульности в конструкциях языка, и физической раскладке модулей по файлам для эффективной раздельной компиляции (главы 8, 9).

2.4.2. Обработка исключений

Когда программа выполнена в виде набора модулей, обработка ошибок должна рассматриваться в контексте модулей. Какой модуль отвечает за обработку тех или иных ошибок? Часто модуль, столкнувшийся с ошибочной ситуацией, не знает, что нужно предпринять для ее исправления. Ответные действия зависят от модуля, инициировавшего операцию, приведшую к ошибке, а не от модуля, выполнявшего эти действия. По мере роста размера программ и в связи с интенсивным использованием библиотек, стандарты по обработке ошибок (или шире — «исключительных ситуаций») становятся важными.

Снова рассмотрим пример с модулем ***Stack***. Что следует предпринять в ответ на попытку занести в стек слишком много символов? Автор модуля ***Stack*** не знает, что нужно пользователю в этом случае, а пользователь не может распознать ситуацию переполнения стека (в противном случае он ее вообще не допустил бы). Устранить указанное противоречие можно следующим образом: автор модуля ***Stack*** выявляет переполнение стека и сообщает об этом (неизвестному) пользователю. В ответ пользователь предпринимает необходимые действия. Например:

```
namespace Stack         // интерфейс
{
  void push (char) ;
  char pop () ;
  class Overflow { } ;  // тип, представляющий исключение, связанное с переполнением
}
```

Обнаружив переполнение, функция ***Stack::push***() может косвенно вызвать код обработки данной исключительной ситуации, сгенерировав «исключение типа ***Overflow***»:

```
void Stack::push (char c)
{
  if (top == max_size) throw Overflow () ;
  // поместить с в стек
}
```

Оператор ***throw*** передает управление обработчику исключений типа ***Stack::Overflow***, расположенному в теле функции, прямо или косвенно вызвавшей функцию ***Stack::push***(). Для этого компилятор *самостоятельно* создает машинный код, *раскручивающий стек вызовов функций назад* до состояния, актуального на момент работы указанной функции. Таким образом, оператор ***throw*** работает как многоуровневый оператор ***return***. Вот соответствующий пример:

```
void f()
{
  // ...
  try  // возникающие здесь исключения передаются обработчику, определенному ниже
  {
    while (true) Stack::push ('c');
  }
  catch (Stack::Overflow)
  {
      // oops: переполнение стека; предпримем надлежащие действия
  }
  // ...
}
```

Здесь цикл *while* в *try-блоке* пытается выполняться вечно. В результате, при некотором обращении к функции *Stack::push()* последняя *сгенерирует исключение* типа *Stack::Overflow*, и управление будет передано в *catch-блок*, предназначенный для обработки исключений этого типа.

Использование механизма исключительных ситуаций языка C++ позволяет сделать обработку ошибок более регулярной, а тексты программ более «читабельными». Более подробно этот вопрос рассматривается в §8.3, главе 14 и приложении Е.

2.5. Абстракция данных

Модульность является фундаментальным аспектом успешных программных проектов большого размера. В настоящей книге она всегда в центре внимания при обсуждении вопросов проектирования программ. Однако способ объявления и использования модулей, рассмотренный нами до сих пор, не является адекватным в случае сложных программных систем. Сейчас я рассмотрю применение модулей для логического формирования того, что можно назвать пользовательскими типами данных, а затем преодолею связанные с таким подходом недостатки с помощью специальных синтаксических конструкций языка C++, предназначенных для явного создания пользовательских типов.

2.5.1. Модули, определяющие типы

Модульное программирование естественным образом приводит к идее о централизованном управлении всеми данными одного и того же типа с помощью специального управляющего модуля. Например, чтобы работать одновременно со множеством стеков, а не с единственным, представленным выше модулем Stack, можно определить одноименный менеджер стеков с интерфейсом следующего вида:

```
namespace Stack
{
  struct Rep;                 // определение раскладки стека находится в другом месте
  typedef Rep& stack;

  stack create();             // создать новый стек
  void destroy(stack s);      // удалить стек s
```

```
  void push (stack s, char c) ;   // поместить c в s
  char pop (stack s) ;            // извлечь s из стека (pop s)
}
```

Объявление

```
struct Rep;
```

говорит о том, что ***Rep*** это имя типа, определение которого будет дано позже (§5.7). Объявление

```
typedef Rep& stack;
```

дает имя stack «ссылкам на ***Rep***» (подробности см. в §5.5). Центральная идея состоит в том, что конкретный стек создается как переменная типа ***Stack*::*stack***, а остальные детали от пользователя скрыты.

Переменные типа ***Stack*::*stack*** ведут себя почти как переменные встроенных типов:

```
struct Bad_pop { };

void f()
{
  Stack::stack s1 = Stack::create () ;     // создаем новый стек
  Stack::stack s2 = Stack::create () ;     // создаем еще один новый стек

  Stack::push (s1, 'c') ;
  Stack::push (s2, 'k') ;

  if(Stack::pop (s1) != 'c') throw Bad_pop () ;
  if(Stack::pop (s2) != 'k') throw Bad_pop () ;

  Stack::destroy (s1) ;
  Stack::destroy (s2) ;
}
```

Реализацию модуля ***Stack*** можно выполнить разными способами. Важно, что пользователю нет необходимости знать, как именно мы это делаем: при неизменном интерфейсе изменение реализации не влияет на код пользователя.

Реализация модуля ***Stack*** может заранее разместить в памяти несколько стеков, и тогда функция ***Stack*::*create*()** будет просто возвращать ссылку на неиспользуемый стек. Функция ***Stack*::*destroy*()**, в свою очередь, будет помечать стек как «свободный», так что функция ***Stack*::*create*()** может использовать его повторно:

```
namespace Stack        // представление
{
  const int max_size= 200;

  struct Rep
  {
    char v[max_size] ;
    int top;
  };

  const int max = 16;   // максимальное количество стеков
```

```
    Rep stacks[max];        // априорная раскладка стеков
    bool used[max];         // used[i] == true если stacks[i] используется

    typedef Rep& stack;
}

void Stack::push(stack s, char c) { /* проверить s на переполнение и push c */ }
char Stack::pop(stack s) { /* проверить, не пуст ли s и pop */ }

Stack::stack Stack::create()
{
    // выбрать неиспользуемый Rep, пометить как используемый,
    // инициализировать его, и вернуть ссылку на него
}

void Stack::destroy(stack s) { /* пометить s как неиспользуемый */ }
```

Что мы здесь сделали по существу? Мы полностью определили интерфейсные функции, используя тип stack из реализации модуля ***Stack***. Результирующее поведение созданного таким образом «стекового псевдотипа» зависит от нескольких факторов: от того, как именно мы определили интерфейсные функции; от того, как мы предоставляем тип ***Stack::stack*** пользователям, а также от деталей определения типов ***Stack::stack*** и ***Stack::Rep*** (так называемых типов представления — representation types).

Рассмотренное решение далеко не идеальное. Существенной проблемой является зависимость способа представления таких псевдотипов пользователю от деталей устройства внутренних типов представления, а ведь пользователи от таких деталей должны быть изолированы. Действительно, стоит нам применить более сложные структуры для внутреннего представления стеков, как сразу же изменятся правила присваивания и инициализации для ***Stack::stack***. Может быть, иногда это и неплохо. Однако ж ясно, что проблема создания удобного типа для стеков просто переместилась в тип представления ***Stack::stack***.

Более фундаментально, пользовательские типы, реализованные посредством модуля с доступом ко внутренней реализации, ведут себя не как встроенные типы и имеют меньшую поддержку. Например, время доступа к типу представления ***Stack::Rep*** контролируется функциями ***Stack::create***`()` и ***Stack::destroy***`()`, а не обычными правилами языка.

2.5.2. Типы, определяемые пользователем

Для преодоления рассмотренной выше проблемы язык C++ позволяет пользователю создавать новые типы, которые ведут себя (почти) так же, как типы встроенные. Их часто называют *абстрактными типами данных*. Однако я предпочитаю называть их *типами, определяемыми пользователем* (*user-defined types*). Строгое определение абстрактных типов данных потребовало бы математически точной «абстрактной» формулировки. При ее наличии то, что мы называем *типами*, было бы конкретными примерами таких истинно абстрактных сущностей. Программная парадигма становится такой:

Решите, какие типы нужны;
обеспечьте полный набор операций для каждого типа.

Если не требуется создавать более одного объекта определенного типа, то достаточно рассмотренного стиля программирования с сокрытием данных на базе модулей.

Математические типы, такие как рациональные или комплексные числа, являются естественными примерами типов, определяемых пользователем. Вот иллюстрация на эту тему:

```
class complex
{
  double re, im;

public:
  complex(double r, double i) { re=r; im=i; }     // создать complex из двух скаляров
  complex(double r) { re=r; im=0; }               // создать complex из скаляра
  complex() { re = im = 0; }                      // по умолчанию: (0,0)

  friend complex operator+ (complex, complex);
  friend complex operator- (complex, complex);    // бинарная операция
  friend complex operator- (complex);             // унарная операция
  friend complex operator* (complex, complex);
  friend complex operator/ (complex, complex);

  friend bool operator== (complex, complex);      // проверка на равенство
  friend bool operator!= (complex, complex);      // проверка на неравенство
  // ...
};
```

Объявление *класса* ***complex*** (то есть типа, определяемого пользователем) задает как представление комплексных чисел, так и набор операций над комплексными числами. При этом представление является *закрытым* (*private*); доступ к полям ***re*** и ***im*** имеют лишь функции, перечисленные в объявлении класса ***complex***. Их можно определить примерно так:

```
complex operator+ (complex a1, complex a2)
{
  return complex(a1.re+a2.re, a1.im+a2.im);
}
```

Функции с именем, совпадающим с именем класса, называют *конструкторами*. Конструктор задает способ инициализации объекта класса. Класс ***complex*** объявляет три конструктора. Один из них преобразует вещественное число типа ***double*** в *комплексное число*, другой создает комплексное число из пары вещественных, а третий строит комплексное число со значением по умолчанию.

Использовать класс complex можно следующим образом:

```
void f(complex z)
{
  complex a = 2.3;
  complex b = 1/a;
  complex c = a+b*complex(1,2.3);
  // ...
  if(c!= b) c = -(b/a)+2*b;
};
```

Компилятор преобразует выполняемые над комплексными числами *операции* в *вызовы соответствующих функций* из определения класса ***complex***. К примеру, для выражения ***c != b*** вызывается функция ***operator!=(c,b)***, а вместо выражения ***1/a*** вызывается функция ***operator/(complex(1),a)***.

Большинства модулей, хотя и не все из них, лучше формулируются в виде типов, определяемых пользователем.

2.5.3. Конкретные типы

Типы, определяемые пользователем, разрабатываются с разными целями. Рассмотрим определяемый пользователем тип ***Stack*** по аналогии с рассмотренным выше типом ***complex***. Пример станет более реалистичным, если мы будем передавать стеку в момент его создания аргумент, фиксирующий предельную емкость (размер) стека:

```
class Stack
{
  char* v;
  int top;
  int max_size;

public:
  class Underflow { };   // используется как исключение
  class Overflow { };    // используется как исключение
  class Bad_size { };    // используется как исключение

  Stack(int s); // конструктор
  ~Stack();     // деструктор

  void push(char c);
  char pop();
};
```

Конструктор ***Stack(int)*** *вызывается автоматически*, когда создается объект класса. Он берет на себя заботу об инициализации. Если требуется очистка памяти (cleanup) в момент выхода объекта из области видимости, можно объявить функцию, называемую *деструктором* (*destructor*; по цели своих действий противоположна конструктору):

```
Stack::Stack(int s)                          // конструктор
{
  top = 0;
  if( s<0 || 10000<s ) throw Bad_size();     // || означает ИЛИ
  max_size = s;
  v = new char[s]; // расположить элементы в свободной памяти (куча, динам. память)
}

Stack::~Stack()     // деструктор
{
  delete[] v;       // уничтожить элементы для дальнейшего
                    // использования памяти (§6.2.6)
}
```

Конструктор инициализирует вновь создаваемый объект класса ***Stack***. Для этого он выделяет некоторое количество памяти из свободного пула (называемого *кучей* или *динамической памятью* — *heap* от *dynamic store*) с помощью операции ***new***. Деструктор же освобождает эту память. Пользователи класса ***Stack*** не предпринимают для этого никаких специальных мер. Они лишь создают объекты типа ***Stack*** и используют их приблизительно так же, как и переменные встроенных типов. Например:

```
Stack s_var1(10);                    // глобальный стек из 10 элементов

void f(Stack& s_ref, int i)          // ссылка на Stack
{
  Stack s_var2(i);                   // локальный стек из i элементов
  Stack* s_ptr= new Stack(20);       // указатель на Stack в свободной памяти

  s_var1.push('a');                  // доступ через переменную (§5.7)
  s_var2.push('b');
  s_ref.push('c');                   // доступ по ссылке (§5.5, §5.7)
  s_ptr->push('d');                  // доступ по указателю (§5.7)
  // ...
}
```

Для типа ***Stack*** действуют те же правила именования, области видимости, выделения памяти, времени жизни и т.д., что и для встроенных типов, таких как, например, ***int*** или ***char***. Подробно о том, как можно управлять временем жизни классовых объектов, рассказано в §10.4.

Естественно, функции ***push***() и ***pop***(), объявленные в классе ***Stack***, должны быть где-то определены:

```
void Stack::push(char c)
{
  if(top == max_size) throw Overflow();
  v[top] = c;
  top = top + 1;
}

char Stack::pop()
{
  if(top == 0) throw Underflow();
  top = top - 1;
  return v[top];
}
```

Такие типы, как ***complex*** или ***Stack***, называют *конкретными типами* (*concrete types*), в противовес *абстрактным типам* (*abstract types*), интерфейс которых еще сильнее изолирует пользователя от деталей реализации.

2.5.4. Абстрактные типы

В процессе перехода от реализованного в виде модуля псевдотипа ***Stack*** (§2.5.1) к соответствующему правильному типу (§2.5.3), одно свойство было утеряно. Мы не отделили представление от пользовательского интерфейса; скорее оно является частью того, что включается в программные фрагменты, использующие тип ***Stack***.

Представление является закрытым, так что к нему могут обращаться лишь объявленные в классе функции. Но оно все же присутствует в явном виде и в том же самом месте. Поэтому, если представление хоть сколько-нибудь значительно изменится, пользовательский код подлежит перекомпиляции. Это плата за то, что пользовательские типы ведут себя так же, как встроенные типы. В частности, у нас не будет настоящих локальных переменных пользовательских типов, если не будет в точности известен размер их представления.

Для редко меняющихся пользовательских типов, и в случаях, когда локальные переменные обеспечивают необходимую ясность и эффективность, все это допустимо, если не идеально. Однако если же мы хотим полностью изолировать пользователей от изменений внутренних реализаций стека, предыдущее определение типа ***Stack*** не годится. В таком случае приходится идти на полное отделение интерфейса от представления (реализации) и на отказ от настоящих локальных переменных этого типа.

Сначала определяем *интерфейс* (*interface*):

```
class Stack
{
public:
  class Underflow { };          // используется как исключение
  class Overflow { };           // используется как исключение
  virtual void push (char c) = 0;
  virtual char pop () = 0;
};
```

Слово ***virtual*** в языках Simula и C++ означает «позднее может быть замещено (переопределено — redefined) в классах, производных от данного». Производный класс призван обеспечить реализацию интерфейса базового класса ***Stack***. Необычный синтаксис ***=0*** означает, что производный класс *обязан* определить эту функцию. Таким образом, класс ***Stack*** служит интерфейсом для любого производного от него класса, реализующего функции ***push***() и ***pop***().

Класс ***Stack*** может быть использован следующим образом:

```
void f(Stack& s_ref)
{
  s_ref.push('c');
  if(s_ref.pop() != 'c') throw Bad_pop();
};
```

Обратите внимание на то, как пользовательская функция ***f***() использует интерфейс класса ***Stack*** при полном неведении относительно деталей реализации. Класс, обеспечивающий интерфейс для множества иных классов, часто называют *полиморфным типом* (*polymorphic type*).

Неудивительно, что реализация может содержать все то из рассмотренного в предыдущем разделе конкретного типа ***Stack***, что осталось за бортом интерфейсного класса ***Stack***:

```
class Array_stack : public Stack  // Array_stack реализует Stack
{
  char* p;
  int max_size;
  int top;
```

```
public:
  Array_stack (int s) ;
  ~Array_stack () ;

  void push (char c) ;
  char pop () ;
};
```

Здесь «:*public*» можно читать как «производный от», «предоставляет реализацию для» или «является подтипом».

Чтобы функция вроде *f*() могла использовать тип ***Stack*** при полном игнорировании деталей реализации, некоторая другая пользовательская функция должна создать объект, с которым функция *f*() и будет реально работать:

```
void g ()
{
  Array_stack as (200) ;
  f(as) ;
}
```

Так как функция *f*() ничего не знает о классе ***Array_stack***, а лишь об интерфейсе ***Stack***, то она будет столь же успешно работать и с иными реализациями интерфейса ***Stack***. Например:

```
class List_stack : public Stack    // List_stack реализует Stack
{
  list<char> lc;                   // (стандартная биб-ка) список символов (§3.7.3)

public:
  List_stack () { }
  void push (char c) { lc.push_front (c) ; }
  char pop () ;
};

char List_stack::pop ()
{
  char x = lc.front () ;           // получить первый элемент
  lc.pop_front () ;                // удалить первый элемент
  return x;
}
```

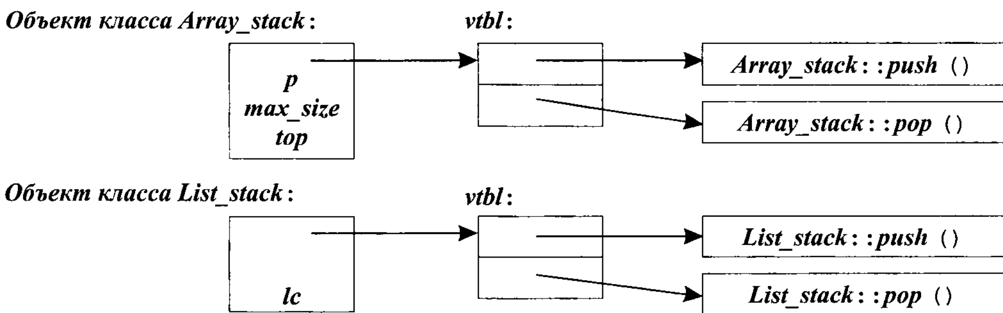
Данное представление базируется на списке символов. Вызов ***lc.push_front***(***c***) добавляет с в качестве первого элемента списка ***lc***, вызов ***lc.pop_front***() удаляет первый элемент, а ***lc.front***() возвращает значение первого элемента списка.

Любая функция может создать объект класса ***List_stack*** и передать его функции *f*():

```
void h ()
{
  List_stack ls;
  f(ls) ;
}
```

2.5.5. Виртуальные функции

Зададимся вопросом, как же вызов ***s_ref.pop***() в теле функции ***f***() связывается с надлежащим определением функции? Ведь когда функция ***f***() вызывается из функции ***h***(), должна работать функция ***List_stack::pop***(). А когда ***f***() вызывается из ***g***(), должна работать иная функция — функция ***Array_stack::pop***(). Для правильного разрешения вызова функций объекты классов, производных от ***Stack***, должны содержать дополнительную информацию, которая прямо указывает, какая функция должна быть вызвана во время выполнения программы. Обычно, компиляторы транслируют имя виртуальной функции в соответствующий индекс в *таблице, содержащей указатели на функции*. Такую таблицу принято называть «*таблицей виртуальных функций*», или просто ***vtbl***. Каждый класс с виртуальными функциями имеет свою собственную таблицу ***vtbl***, идентифицирующую эти функции. Графически это можно отобразить следующим образом:



Таблицы виртуальных функций позволяют корректно использовать объекты даже в тех случаях, когда на момент вызова функции неизвестны ни размеры объекта, ни местоположение его данных. Единственное, что нужно знать вызывающей стороне — это расположение таблицы ***vtbl*** класса и индексы в этой таблице для вызова виртуальных функций. Механизм вызова виртуальных функций по сути столь же эффективен, как и вызов обычных функций. Дополнительные затраты памяти сводятся к одному указателю на каждый объект класса с виртуальными функциями и на таблицу ***vtbl*** для самого класса.

2.6. Объектно-ориентированное программирование

Абстракция данных чрезвычайно важна для выполнения качественного проектирования и она будет в центре нашего внимания на протяжении всей книги. Однако типы, определяемые пользователем, сами по себе не настолько гибкие, чтобы удовлетворить все наши потребности. В настоящем разделе сначала демонстрируется проблема, связанная с простым пользовательским типом, а затем показывается, как ее преодолеть с помощью классовых иерархий.

2.6.1. Проблемы, связанные с конкретными типами

Конкретный тип, вроде «псевдотипа», реализованного в виде модуля, представляется этаким черным ящиком. Будучи определенным, черный ящик практически не взаимодействует с остальными частями программы. Приспособить его для других целей можно лишь, модифицировав его определение. С одной стороны, можно считать такое свойство конкретных типов как идеальное, а с другой стороны — это источник чрезвычайной негибкости. Рассмотрим сейчас определение типа ***Shape***, который будет использоваться в графической системе. Для начала считаем, что графическая система должна поддерживать окружности (circles), треугольники (triangles) и квадраты (squares). Кроме того, предполагаем, что у нас есть

```
class Point { /* ... */ };
class Color { /* ... */ };
```

Здесь /* и */ маркируют начало и конец комментариев. Такая нотация подходит и для многострочных комментариев, и для комментариев, оканчивающихся где-то в середине строки (так что после них в той же строке располагается незакомментированный код).

Мы можем определить тип ***Shape*** приблизительно так:

```
enum Kind {circle, triangle, square };    // перечисление (§4.8)

class Shape
{
  Kind k;                                  // поле типа
  Point center;
  Color col;
  // ...

public:
  void draw () ;
  void rotate (int) ;
  // ...
};
```

Поле ***k***, которое можно назвать *полем типа* (*type field*), требуется, чтобы операции, вроде ***draw***() или ***rotate***(), могли установить, с каким типом фигуры они имеют дело. При этом код функции ***draw***() может быть устроен следующим образом:

```
void Shape::draw ()
{
  switch (k)
  {
    case circle:
      // рисуем окружность
      break;
    case triangle:
      // рисуем треугольник
      break;
    case square:
      // рисуем квадрат
      break;
  }
}
```

Это просто несъедобное варево. Получается, что функции вроде ***draw***(), обязаны знать обо всех типах фигур, с которыми система имеет дело. Код таких функций растет по мере добавления в систему новых видов обрабатываемых фигур. Каждый раз, когда мы добавляем новую фигуру, нужно тщательно изучить код каждой операции и (возможно) изменить его. Если у нас нет доступа к исходному коду каждой операции, то мы вообще не можем добавлять новые фигуры. Поскольку добавление новых фигур означает вторжение в код важных операций, оно требует мастерства и потенциально чревато внесением ошибок в код, обрабатывающий старые фигуры. Выбор представления для некоторых фигур может оказаться слишком тесным, если потребуется втиснуть его в типично жесткие рамки представления общего типа ***Shape***.

2.6.2. Классовые иерархии

Проблема заключается в том, что не видно разницы между *общими свойствами* всех фигур (любая фигура обладает определенным цветом, может быть нарисована и т.д.) и *особыми свойствами* отдельных их типов (окружность характеризуется радиусом и рисуется специфическим способом). *Объектно-ориентированное программирование* призвано выявлять такую разницу и извлекать из этого максимальную пользу. Языки, которые располагают средствами, позволяющими явно выразить указанную разницу и использовать этот факт, поддерживают объектно-ориентированное программирование. А остальные языки — нет.

Механизм наследования (заимствованный в C++ из Simula) решает проблему. Сначала мы определяем класс, который задает общие свойства всех фигур:

```
class Shape
{
  Point center;
  Color col;
  // ...

public:
  Point where() {return center; }
  void move(Point to) {center = to; /* ... */ draw(); }

  virtual void draw() = 0;
  virtual void rotate(int angle) = 0;
  // ...
};
```

Аналогично рассмотренному в §2.5.4 абстрактному типу ***Stack***, функции, для которых интерфейс вызова может быть сформулирован, а реализация — нет, объявляются *виртуальными*. В частности, функции ***draw***() и ***rotate***() могут определяться лишь для конкретных типов фигур. Поэтому они и объявлены виртуальными.

Отталкиваясь от определения класса ***Shape***, мы можем писать *универсальные* функции, которые манипулируют векторами указателей на фигуры:

```
void rotate_all(vector<Shape*>& v, int angle)    // поворачиваем элементы v
                                                  // на angle градусов
{
  for(int i = 0; i<v.size(); ++i) v[i]->rotate(angle);
}
```

Чтобы задать конкретную фигуру, мы должны, во-первых, указать, что это фигура вообще, а во-вторых, определить дополнительную специфику (включая виртуальные функции):

```
class Circle : public Shape
{
  int radius;

public:
  void draw() { /* ... */ }
  void rotate(in) { } // да, функция ничего не делает
};
```

В языке C++ говорят, что класс **Circle** *наследуется от* (*is derived from*) класса **Shape**, а класс **Base** называется *базовым* (*base*) для класса **Circle**. Альтернативная терминология называет **Circle** и **Shape** *подклассом* и *суперклассом* (*subclass* и *superclass*), соответственно. Производный класс наследует все члены базового класса, а потому эту тему принято называть *наследованием* (*inheritance*) классов. Новая парадигма программирования звучит так:

Решите, какие классы нужны;
обеспечьте полный набор операций для каждого класса;
явно выразите общность посредством наследования.

Там, где нет общности типов, достаточно одной лишь абстракции данных. Степень общности между типами, которую можно выявить и отразить с помощью наследования и виртуальных функций, является лакмусовой бумажкой для определения того, насколько применим объектно-ориентированный подход к данной задаче. В некоторых предметных областях, таких как интерактивная графика, объектно-ориентированный подход применим широчайшим образом. В других предметных областях, например в области математических вычислений, достаточно абстракции данных, а применение средств объектно-ориентированного программирования представляется ненужным или избыточным.

Выявление общности между типами не является тривиальной задачей. Степень общности, которую можно найти, зависит от способа проектирования программной системы. Начинать искать общность типов нужно уже на начальной стадии проектирования, когда имеются лишь технические требования к системе. Классы могут специально проектироваться как строительные блоки для других классов. Уже существующие классы можно подвергнуть анализу на предмет выявления у них общих черт, которые можно выделить в общий базовый класс.

Попытки рассмотреть вопрос о том, что такое объектно-ориентированное программирование, не обращаясь при этом к конкретным конструкциям языков программирования, предприняты в работах [Kerr,1987], и [Booch,1994] (см. §23.6).

Иерархии классов и абстрактные классы (§2.5.4) дополняют, а не исключают друг друга (§12.5). Вообще, перечисленные здесь парадигмы имеют тенденцию к комплиментарности и взаимоподдержке. Например, и классы, и модули содержат функции, в то время как модули содержат и классы, и функции. Опытный разработчик применяет различные парадигмы по мере необходимости.

2.7. Обобщенное программирование

Тому, кто захочет использовать стек, вряд ли всегда будет требоваться именно стек символов. Стек это более общее понятие, не зависящее от символов. Соответственно, его желательно и определять независимым образом.

Вообще, если алгоритм можно выразить независимо от деталей его реализации, причем приемлемым образом и без логических искажений, то так и надо делать.

Соответствующая программная парадигма звучит так:

Решите, какие нужны алгоритмы;
параметризуйте их так, чтобы они могли работать
со множеством подходящих типов и структур данных.

2.7.1. Контейнеры

Мы можем *обобщить* стек символов до *стека элементов любого типа*, если объявим его *шаблоном*, а тип элементов ***char*** заменим на *параметр шаблона*. Вот соответствующий пример:

```
template<class T> class Stack
{
  T* v;
  int max_size;
  int top;

public:
  class Underflow { };
  class Overflow { };

  Stack (int s) ;          // конструктор
  -Stack ( ) ;             // деструктор

  void push (T) ;
  T pop ( ) ;
};
```

Префикс ***template<class T>*** сообщает следующему за ним объявлению, что ***T*** — это *параметр типа.*

Функции-члены определяются аналогичным образом:

```
template<class T> void Stack<T>: :push (T c)
{
  if(top == max_size) throw Overflow ( ) ;
  v[top] = c;
  top = top + 1;
}

template<class T> T Stack<T>: :pop ( )
{
  if(top == 0) throw Underflow ( ) ;
  top = top - 1;
  return v[top] ;
}
```

Теперь приведем пример кода, использующего новое определение стека:

```
Stack<char> sc(200);              // стек для 200 символов
Stack<complex> scplx(30);         // стек для 30 комплексных чисел
Stack< list<int> > sli(45);       // стек для 45 списков целых

void f()
{
  sc.push('c');
  if(sc.pop() != 'c') throw Bad_pop();

  scplx.push(complex(1,2));
  if(scplx.pop() != complex(1,2)) throw Bad_pop();
}
```

С помощью шаблонов можно также определять списки, вектора, ассоциативные массивы и т.д. Классы, предназначенные для хранения набора элементов некоторого типа, принято называть *контейнерными классами* (*container classes*), или просто *контейнерами*.

Механизм шаблонов автоматически обрабатывается на стадии компиляции программы и не вносит дополнительных накладных расходов на стадии ее выполнения (что типично для ручного кодирования).

2.7.2. Обобщенные алгоритмы

Стандартная библиотека языка C++ содержит богатый набор контейнеров, кроме того, пользователи могут разрабатывать собственные контейнеры (главы 3, 17, 18). В результате мы сталкиваемся с ситуацией, когда желательно еще раз применить парадигму обобщенного программирования и *параметризовать алгоритмы по типам контейнеров*. Например, мы хотим сортировать, копировать элементы и осуществлять поиск в векторах, списках и массивах, но не хотим писать отдельные варианты функций ***sort***(), ***copy***() и ***search***() для каждого из типов контейнеров. Кроме того, вряд ли мы захотим жестко фиксировать структуру данных, принимаемую единственными вариантами функций. Поэтому возникает задача обобщенного доступа к элементам контейнера, который не зависит от типа обрабатываемого контейнера.

Один из подходов, а именно подход к контейнерам и нечисловым алгоритмам, принятый в стандартной библиотеке языка C++ (§3.8, глава 18), фокусируется на понятии *последовательностей элементов* (*sequences*) и на манипулировании ими с помощью *итераторов* (*iterators*).

Понятие последовательности элементов графически иллюстрируется следующим образом:

У последовательности элементов (elements) есть начало (begin) и конец (end). Итератор ссылается на элемент последовательности, и для него существует операция, позволяющая ему перейти к следующему элементу последовательности. Конец последовательности представляется итератором, ссылающимся на нечто, расположенное за последним элементом последовательности. Физическое представление

«конца» жестко не фиксируется; им может быть специальный «*элемент-часовой*» (*sentinel element*), или что-то другое. Рассмотренное понятие последовательности элементов одинаково применимо и к спискам, и к массивам.

Нам требуются стандартные обозначения для операций «доступ к элементам посредством итератора» и «перевод итератора к следующему элементу». Очевидным выбором (при понимании общей идеи) является выбор стандартных знаков операции * (*операция разыменования — dereference operator*) и операции ++ (*операция инкремента — increment operator*), соответственно.

Теперь мы можем писать следующий код:

```
template<class In, class Out> void copy(In from, In too_far, Out to)
{
  while (from != too_far)
  {
    *to = *from;         // копируем элемент указуемый итератором
    ++to;                // следующий выходной элемент
    ++from;              // следующий входной элемент
  }
}
```

Этот *обобщенный код* копирует элементы *любых контейнеров*, для которых можно определить итераторы с правильным синтаксисом и семантикой.

Наш обобщенный копирующий алгоритм можно также применить и ко встроенным в C++ низкоуровневым массивам и указателям, поскольку требующиеся операции для них определены:

```
char vc1[200];           // массив из 200 символов
char vc2[500];           // массив из 500 символов

void f()
{
  copy(&vc1[0], &vc1[200], &vc2[0]);
}
```

Здесь все элементы массива **vc1** копируются в начало массива **vc2**.

Все контейнеры стандартной библиотеки (§16.3, глава 17) поддерживают концепции последовательностей и итераторов.

У нас в шаблоне используются два параметра типа, а не один; параметр ***In*** относится к типу контейнера-источника, а параметр ***Out*** — к типу контейнера приемника. Это позволяет копировать элементы контейнера одного типа в контейнер другого типа. Например:

```
complex ac[200];

void g(vector<complex>& vc, list<complex>& lc)
{
  copy(&ac[0], &ac[200], lc.begin());
  copy(lc.begin(), lc.end(), vc.begin());
}
```

Этот код копирует массив в контейнер типа ***list***, а затем последний копируется в контейнер типа ***vector***. Для любого стандартного контейнера функциональный вызов ***begin***() возвращает итератор, указывающий на первый элемент последовательности.

2.8. Заключение

Не существует идеальных языков программирования. К счастью, язык и не обязан быть идеальным, чтобы быть подходящим инструментом для создания больших программных систем. На самом деле, язык программирования общего назначения не может быть идеальным одновременно для всех решаемых с его помощью задач. То, что идеально для одной задачи, в другой является скорее недостатком, ибо достижение совершенства предполагает специализацию. Язык C++ разрабатывался так, чтобы он служил хорошим инструментом для решения широкого круга задач, и чтобы на нем можно было явно выразить широкий круг идей (идиом программирования).

Не все можно выразить непосредственно, пользуясь встроенными средствами языка. Но это и не нужно. Средства языка существуют для поддержки разнообразных технологий и стилей программирования. Как следствие, в процессе изучения языка нужно фокусироваться на освоении стилей программирования, для него родных и естественных, а вовсе не на подробнейшем изучении мельчайших деталей языковых конструкций.

В практическом программировании мало толку от знания «потайных мест» языка или от применения максимально возможных средств. Отдельные средства языка сами по себе не представляют большого интереса. Только во взаимодействии с другими средствами и в контексте применяемой технологии они приобретают смысл и интерес. Так что, читая последующие главы, помните, что истинной целью изучения деталей языка является стремление научиться использовать их так, чтобы поддерживать выбранный стиль программирования в контексте основательного плана построения программной системы.

2.9. Советы

1. Не паникуйте раньше времени! Все со временем станет понятным; §2.1.
2. Для написания хороших программ вы не обязаны знать каждую деталь языка C++; §1.7.
3. Сосредоточьте внимание на технологиях программирования, а не на деталях языка; §2.1.

3

Обзор стандартной библиотеки

Зачем тратить время на обучение,
если невежество достигается мгновенно?
— Хоббс

https://t.me/it_boooks

Стандартная библиотека — вывод — строки — ввод — вектора — проверка диапазона — списки — ассоциативные массивы — обзор контейнеров — алгоритмы — итераторы — итераторы ввода/вывода — предикаты — алгоритмы, использующие функции-члены классов — обзор алгоритмов — комплексные числа — векторная арифметика — обзор стандартной библиотеки — советы.

3.1. Введение

Ни одна значительная программа не написана исключительно лишь на «голых» конструкциях языка программирования. Сначала разрабатываются библиотеки поддержки. Затем они становятся основой для дальнейшей работы.

В продолжении главы 2 здесь дается краткий обзор основных библиотечных средств, дающий представление о том, что в итоге может быть создано с помощью языка C++ и его стандартной библиотеки. Рассматриваются такие полезные типы стандартной библиотеки, как ***string***, ***vector***, ***list*** и ***map***, а также наиболее употребительные варианты их использования. Это позволяет мне приводить более осмысленные примеры и давать более интересные упражнения в последующих главах книги. Как и в главе 2, здесь также настоятельно рекомендуется не отвлекаться и не огорчаться из-за неполного понимания деталей излагаемого материала. Цель настоящей главы состоит в том, чтобы дать вам почувствовать, с чем придется столкнуться при дальнейшем обучении, а также помочь овладеть простейшими способами использования наиболее полезных средств библиотеки. Подробное введение в стандартную библиотеку дано в §16.1.2.

Элементы стандартной библиотеки, которые изучаются в настоящей книге, входят составной частью в любое достаточно полное средство разработки программ на языке C++ (именуемое также средой разработки). В дополнение к стандартной библиотеке

языка C++ эти среды часто содержат средства реализации графического интерфейса пользователя (оконных систем или GUI — Graphical User Interface), предназначенного для удобного визуального взаимодействия пользователя с работающей программой. Кроме того, многие среды поставляются еще и с разного рода «фундаментальными библиотеками», призванными поддержать корпоративные или промышленные стандарты разработки и/или выполнения программ. Я не рассматриваю такие дополнительные средства и библиотеки. Я просто хочу дать полное описание языка C++ в соответствии со стандартом, а также обеспечить переносимость всех примеров кода, за исключением особо оговариваемых случаев. Естественно, программист может сам, в качестве самостоятельных упражнений, ознакомиться и овладеть дополнительными средствами и библиотеками, поставляемыми с конкретной средой разработки.

3.2. Hello, world! (Здравствуй, мир!)

Вот пример минимальной программы на языке C++:

```
int main () { }
```

Она определяет функцию ***main*** (), не принимающую аргументов и ничего не делающую.

Каждая программа на C++ должна иметь функцию с именем ***main***. Это точка входа в программу, с которой и начинается ее выполнение. Значение типа ***int***, возвращаемое программой (если оно вообще имеется, конечно), предназначено для системы, запустившей программу. Если возврат отсутствует, то система получит сообщение об успешном завершении программы. Ненулевой возврат означает аварийное завершение.

В типичном случае программы выводят некоторую информацию о своей работе. Например, следующая программа выводит строку ***Hello, world***!:

```
#include <iostream>

int main ()
{
  std::cout << "Hello, world!\n";
}
```

Строка #***include*** <***iostream***> заставляет компилятор *включить* сюда все объявления стандартных потоковых средств ввода-вывода (I/O facilities) из файла ***iostream***. Без этих объявлений выражение

```
std::cout << "Hello, world!\n"
```

не имело бы смысла. Операция << («вставить в поток») осуществляет вывод своего правого аргумента в левый аргумент. В данном случае, строковый литерал "***Hello, world***!***n***» записывается в стандартный поток вывода ***std***::***cout***. *Строковый литерал* (*string literal*) является последовательностью символов, заключенных в двойные кавычки. В строковых литералах символ \ (обратная наклонная черта) и следующий за ним в совокупности означают один специальный символ. В данном случае, ***n*** означает специальный символ «*перевода строки*», так что сначала выводится текст ***Hello***, ***world***!, а затем осуществляется переход на новую строку (на новую строку устройства вывода).

3.3. Пространство имен стандартной библиотеки

Стандартная библиотека определена в пространстве имен ***std*** (§2.4, §8.2). Вот почему я пишу ***std::cout***, а не просто ***cout***. То есть я указываю явным образом, что используется именно ***cout*** из стандартной библиотеки, а не какой-то другой ***cout***.

Любое средство стандартной библиотеки объявлено в каком-либо *стандартном заголовочном файле*, аналогичном файлу ***iostream***. Например:

```
#include <string>
#include <list>
```

Теперь доступны такие типы стандартной библиотеки, как ***string*** и ***list***. Для этого нужно явно использовать префикс ***std::***, как в следующем примере:

```
std::string s = "Four legs Good; two legs Baaad!";
std::list<std::string> slogans;
```

Однако для упрощения записи примеров в дальнейшем я буду редко использовать префикс ***std::***, и также редко буду явно включать в код примеров строки ***#include*** с необходимыми стандартными заголовочными файлами. Чтобы скомпилировать и запустить на выполнение код примеров, вы сами должны включить все необходимые заголовочные файлы (перечислены в §3.7.5, §3.8.6 и §16.1.2). Кроме того, вы должны или явно добавлять префикс ***std::***, или делать все имена из ***std*** глобально доступными (§8.2.3). Например:

```
#include <string>    // включает объявления стандартных средств работы со строками
using namespace std;            // делает имена из std доступными без префикса std::
string s = "Ignorance is bliss!";  // ok: string означает std::string
```

В общем случае, «сброс» имен из ***std*** в одно глобальное пространство имен является дурным тоном. И, тем не менее, чтобы укоротить тексты примеров, описывающих применение средств C++ и библиотеки, я часто опускаю повторяющиеся строки ***#include*** и ***std::*** префиксы. В настоящей книге я использую почти исключительно средства стандартной библиотеки языка C++, так что, если в примере встречается имя из стандартной библиотеки, то это либо демонстрация его стандартного применения, либо фрагмент, поясняющий, как стандартное библиотечное решение может быть определено.

3.4. Вывод

В стандартной библиотеке с помощью заголовочного файла ***iostream*** определяются *средства вывода* для каждого встроенного типа. Кроме того, несложно расширить эти средства и на пользовательские типы. По умолчанию, величины, выводимые в поток ***cout***, автоматически преобразуются в *последовательный набор символов*. Например, следующий код

```
void f()
{
  cout<< 10;
}
```

поместит в стандартный поток вывода сначала символ *1*, а затем символ *0*. То же самое происходит и в следующем примере:

```
void g ()
{
  int i = 10;
  cout<< i;
}
```

Данные разных типов смешиваются очевидным образом:

```
void h (int i)
{
  cout<< "the value of i is ";
  cout<< i;
  cout<< '\n';
}
```

Если i равно 10, то в поток вывода попадет следующая последовательность символов:

the value of i is 10

Символьной константой (*character constant*) называется символ, заключенный в одиночные кавычки. Отметим, что символьные константы выводятся в поток именно как символы, а не как соответствующие им числовые значения. Например, код

```
void k ()
{
  cout<<'a';
  cout<<'b';
  cout<<'c';
}
```

поместит в выходной поток последовательность ***abc***.

Утомительно то и дело повторять имя стандартного потока вывода при последовательном выводе нескольких величин. К счастью, результат операции << можно использовать для последующей операции вывода. Например:

```
void h2 (int i)
{
  cout<< "the value of i is "<<i<<'\n';
}
```

По своему действию эта функция эквивалентна представленной ранее функции ***h***(). Подробнее потоки рассматриваются в главе 21.

3.5. Строки

В стандартной библиотеке определен тип ***string***, дополняющий рассмотренные ранее строковые литералы. Тип ***string*** обеспечивает множество полезных операций, таких как *конкатенация* (*concatenation*). Например:

```
string s1 = "Hello";
string s2 = "world";

void m1()
{
  string s3 = s1+", "+s2+"!\n";
  cout<<s3;
}
```

Здесь строка *s3* инициализируется значением (текстом)

Hello, world!

со специальным знаком «перевод строки» на конце. Сложение строк означает их конкатенацию. Строки можно складывать со строками, строковыми литералами и символами.

Во многих приложениях конкатенация реализуется в форме добавления чего-либо к концу заданной строки, что напрямую поддерживается операцией +=. Например:

```
void m2(string& s1, string& s2)
{
  s1 = s1 + '\n';   // добавить к концу содержимого символ перевода строки
  s2 += '\n';       // добавить к концу содержимого символ перевода строки
}
```

Показанные способы добавления к концу строки семантически эквивалентны, но я предпочитаю второй из них, так как он короче и, к тому же, реализуется эффективнее в машинном коде.

Естественно, что строки можно сравнивать между собой и со строковыми литералами. Например:

```
string incantation;

void respond(const string& answer)
{
  if(answer==incantation)
  {
    // сотворим чудо (incantation - колдовство, заклинание)
  }
  else if(answer=="yes")
  {
    // ...
  }
  // ...
}
```

Класс *string* из стандартной библиотеки описывается в главе 20. Помимо прочих полезных вещей он предоставляет возможность работы с подстроками. Например:

```
string name="Niels Stroustrup";

void m3()
{
  string s = name.substr(6,10);     // s ="Stroustrup"
```

```
  name.replace(0,5,"Nicholas"); // name становится "Nicholas Stroustrup"
}
```

Операция[1] ***substr***() возвращает копию подстроки, задаваемой аргументами. Первый аргумент означает индекс (позицию) начала подстроки, а второй — ее длину. Так как начальным значением индекса является нуль, то строка ***s*** принимает значение ***Stroustrup***.

Операция ***replace***() замещает исходное значение подстроки на новое. В данном случае, подстрока начинается с индекса ***0*** и имеет длину ***5***, так что это ***Niels***; она замещается на ***Nicholas***; таким образом, окончательное значение строки ***name*** есть ***Nicholas Stroustrup***. Дополнительно отметим, что длина замещаемой подстроки и нового значения не обязаны совпадать.

3.5.1. С-строки

Как известно, С-строки (или строки языка С) — это массивы символов с терминальным нулем (§5.2.2). С-строки легко преобразуются в значения типа ***string***. Но может потребоваться и обратное преобразование, например, когда имеется значение типа ***string***, а нужно вызвать функцию, принимающую аргумент типа С-строки. Его-то и нужно тогда извлечь из значения типа ***string*** с помощью функции ***c_str***() (§20.3.7). Например, мы можем вывести значение строки name библиотечной функцией ***printf***() языка С (§21.8):

```
void f()
{
  printf("name: %s\n",name.c_str());
}
```

3.6. Ввод

В стандартной библиотеке *тип* ***istream*** управляет посимвольным вводом встроенных типов данных точно так же, как рассмотренный выше тип ***ostream*** управляет их выводом. Действие типа ***istream*** несложно распространить и на пользовательские типы.

Для обозначения операции ввода используется знак >> («извлечь из потока»), а ***cin*** обозначает *стандартный поток ввода*. Тип правого операнда операции >> определяет, что именно вводится и куда записывается результат ввода. Например, следующий код

```
void f()
{
  int i;
  cin>> i;      // считать целое значение в i
  double d;
```

[1] Здесь, и во многих местах далее, автор использует слово «операция» как синоним слова «действие» в самом широком смысле. В строгом же смысле, здесь имеет место функциональный вызов. — *Прим. ред.*

```
    cin>>d;        // считать значение с плавающей запятой двойной точности в d
}
```

читает число, вроде ***1234***, из потока ввода в целую переменную ***i*** и значение с плавающей запятой, вроде ***12.34e5***, в переменную ***d*** типа ***double*** (числа с плавающей запятой двойной точности).

Ниже дан пример программы, осуществляющей преобразования из дюймов в сантиметры и обратно. Вы вводите число и символ, обозначающий единицу измерения: сантиметры (символ ***'c'***) или дюймы (символ ***'i'***). Программа выведет соответствующее значение в других единицах измерения:

```
int main ()
{
  const float factor = 2.54;      // 1 дюйм равен 2.54 cm
  float x, in, cm;
  char ch = 0;

  cout<<"enter length: ";

  cin>>x;                         // считать число с плавающей запятой
  cin>>ch;                        // считать суффикс

  switch (ch)
  {
    case 'i':                     // дюймы
      in=x;
      cm=x*factor;
      break;
    case 'c'                      // сантиметры
      in=x/factor;
      cm=x;
      break;
    default:
      in=cm=0;
      break;
  }

  cout<<in<<" in= "<<cm<<" cm\n";
}
```

Оператор ***switch*** сравнивает значение с набором констант. Оператор ***break*** осуществляет выход из оператора ***switch***. Константы в разных ***case***-ветвях должны быть разными. Если проверяемое значение не совпадает ни с одной из них, то выбирается ***default***-ветвь. Программист, в общем случае, не обязан реализовывать ***default***-ветвь.

Часто требуется прочитать последовательность символов. В этом случае удобно помещать результат ввода в объект типа ***string***. Вот соответствующий пример:

```
int main ()
{
  string str;

  cout<<"Please, enter your name\n";
  cin >> str;
```

```
  cout<<"Hello, "<<str<<"!\n";
}
```

Если вы введете

Eric

то вывод будет таким:

Hello, Eric!

По умолчанию, символ-разделитель (§5.2.2), например, пробел, прерывает ввод, так что если вы введете

Eric Bloodaxe

притворяясь несчастным королем Йорка, то все равно на выходе получите то же самое:

Hello, Eric!

Считать целиком всю строку можно с помощью функции ***getline*()**, например:

```
int main()
{
  string str;

  cout<<"Please, enter your name\n";
  getline(cin, str);
  cout<<"Hello, "<<str<<"!\n";
}
```

Теперь при вводе

Eric Bloodaxe

на выходе мы получим желаемое:

Hello, Eric Bloodaxe!

Тип string имеет чудесное свойство принимать на хранение тексты любого размера, так что если вы вдруг введете пару мегабайт двоеточий, то программа выведет вам в ответ всю эту кучу знаков — если только компьютер или операционная система не исчерпают при этом какой-либо критический ресурс.

3.7. Контейнеры

Часто в процессе решения самых разных задач удобно создать коллекцию (набор) объектов некоторого типа, чтобы потом такой коллекцией манипулировать. Простейшим примером является процесс чтения символов в строку и последующий вывод строки на дисплей (или принтер). Класс, *основной целью* которого является *хранение объектов*, называется *контейнером*. Выбор подходящих контейнеров и реализация фундаментальных методов работы с ними, достаточных и удобных для решения конкретной задачи, являются важным этапом разработки программ.

Для иллюстрации наиболее важных контейнеров стандартной библиотеки рассмотрим простую программу, хранящую имена и номера телефонов. Это задача,

различные подходы к решению которой хорошо понятны специалистам разных профилей.

3.7.1. Контейнер vector

Для большинства программистов на C встроенный массив пар значений (имя, телефонный номер) является естественной точкой отсчета:

```
struct Entry                // запись телефонной книги
{
  string name;
  int number;
};

Entry phone_book[1000];     // телефонная книга на 1000 записей

void print_entry(int i)     // использование (вывод записи)
{
  cout<<phone_book[i].name<<' '<< phone_book[i].number<<'\n';
}
```

Однако встроенные массивы имеют жестко фиксированные размеры. Если мы выберем избыточный размер, то впустую израсходуем память; если же размер будет недостаточным, то произойдет переполнение. В любом случае, придется писать низкоуровневый код по управлению выделением памяти. Контейнер ***vector*** из стандартной библиотеки (§16.3) берет на себя эту заботу:

```
vector<Entry> phone_book(1000);

void print_entry(int i)     // используем как раньше
{
  cout<<phone_book[i].name <<' '<<phone_book[i].number<<'\n';
}
void add_entries(int n)     // увеличиваем размер книги на n записей
{
  phone_book.resize(phone_book.size()+n);
}
```

Функция ***size()***, член класса ***vector***, возвращает количество размещенных элементов.

Обратите внимание на круглые скобки в определении объекта ***phone_book***. Мы создали единственный объект типа ***vector<Entry>*** и проинициализировали его начальным размером. Это разительно отличается от объявления встроенного массива:

```
vector<Entry> book(1000);    // vector из 1000 элементов
vector<Entry> books[1000];   // 1000 пустых векторов
```

Если в объявлении объекта вы по ошибке используете квадратные скобки вместо круглых, то компилятор наверняка обнаружит ошибку в тот момент, когда вы попытаетесь использовать объект.

Объекты типа ***vector*** могут участвовать в операциях присваивания:

```
void f(vector<Entry>& v)
{
  vector<Entry> v2 = phone_book;
```

```
  v=v2;
  // ...
}
```

При этом копируются все элементы векторов. Таким образом, после инициализации и операции присваивания в функции *f*() векторы *v* и *v2* содержат независимые копии объектов типа *Entry* из телефонной книги. Когда вектора содержат много элементов, невинно выглядящие инициализация и присваивание могут оказаться недопустимо дорогостоящими операциями. В таких случаях лучше использовать ссылки или указатели.

3.7.2. Проверка диапазона индексов

Стандартный тип *vector* по умолчанию никаких проверок индексов не производит (§16.3.3). Например:

```
void f()
{
  int i=phone_book[1001].number;          // 1001 вне диапазона индексов
  // ...
}
```

Вероятнее всего, такое присваивание поместит в переменную *i* некоторое неконтролируемое значение, а не породит сообщение об ошибке. Это нежелательно, и поэтому в следующих разделах я буду использовать контейнер *Vec* — простую адаптацию класса *vector*, проверяющую, попадает ли индекс в допустимый диапазон значений. Тип *Vec* во всем эквивалентен типу *vector*, но он генерирует исключение типа *out_of_range* (определенное в файле *<stdexcept>*), когда индекс выходит за границы допустимого диапазона.

Технологии создания типов, подобных *Vec*, и эффективного использования исключений рассматриваются в §11.12, §8.3, главе 14 и Приложении Е. Однако для иллюстрирующих примеров в настоящей книге вполне достаточно следующего определения:

```
template<class T> class Vec : public vector<T>
{
public:
  Vec() :vector<T>() { }
  Vec(int s) :vector<T>(s) { }

  T& operator[] (int i) {return at(i); }                // с проверкой диапазона
  const T& operator[] (int i) const {return at(i); }    // с проверкой диапазона
};
```

Для доступа к элементам векторов по индексу класс vector определяет функцию *at*(), генерирующую исключение типа *out_of_range* при выходе индекса за границы допустимого диапазона (§16.3.3). При необходимости, доступ к базовому типу *vector<T>* может быть заблокирован; см. §15.3.2.

Возвращаясь к задаче о хранении имен и телефонных номеров, мы можем теперь смело применить класс *Vec*, будучи полностью уверенными в том, что выход индексов за границу допустимого диапазона будет надежно перехвачен. Например:

```
Vec<Entry> phone_book(1000);

void print_entry(int i)                     // используем как раньше
{
  cout<<phone_book[i].name<<' '<<phone_book[i].number<<'\n';
}
```

Выход индекса за границу допустимого диапазона вызовет генерацию исключения, которое пользователь может перехватить. Например:

```
void f()
{
  try
  {
    for(int i=0; i<10000; i++) print_entry(i);
  }
  catch(out_of_range)
  {
    cout<<"range error\n";
  }
}
```

Исключение будет сгенерировано и перехвачено при попытке вычисления выражения ***phone_book***[***i***] в момент, когда выполнится условие ***i==1000***. Если пользователь не перехватит это исключение, программа будет принудительно снята с выполнения стандартным образом, а не продолжит выполнение с непредсказуемыми последствиями. Для минимизации сюрпризов, связанных с исключениями, в качестве тела стартовой функции ***main***() можно использовать ***try***-*блок*:

```
int main()
try
{
  //ваш код
}
catch(out_of_range)
{
  cerr<<"range error\n";
}
catch(...)
{
  cerr<<"unknown exception thrown\n";
}
```

Если мы не перехватим исключение, связанное с выходом за границу допустимого диапазона индексов, или иные исключения, то сообщение об ошибке будет записано в стандартный поток ошибок ***cerr*** (§21.2.1).

3.7.3. Контейнер list

Включение новых абонентов в телефонный справочник и удаление из него ненужных записей являются распространенными операциями. Из-за этого, для телефонного справочника больше подходит *список* (***list***), а не вектор. Например:

```
list<Entry> phone_book;
```

Если мы используем список, то лишаемся доступа к элементам по индексу. Вместо этого, мы осуществляем поиск элемента с заданным значением. При этом мы опираемся на идею о представлении контейнеров в виде последовательности элементов (см. §3.8):

```
void print_entry (const string& s)
{
  typedef list<Entry>::const_iterator LI;

  for (LI i=phone_book.begin() ; i != phone_book.end() ; ++i)
  {
    const Entry& e = *i;          // ссылка используется ради краткости записи
    if(s==e.name)
    {
      cout<<e.name<<' '<<e.number<<'\n';
      return;
    }
  }
}
```

Поиск строки s начинается с начала списка и продолжается либо до ее обнаружения в некотором элементе, либо завершается достижением конца списка. Любой контейнер стандартной библиотеки предоставляет функции ***begin***() и ***end***(), *возвращающие итераторы*, которые настроены, соответственно, на первый элемент и на элемент, следующий «за последним» (§16.3.2). Если итератор ***i*** ссылается на некоторый элемент контейнера, то ++***i*** ссылается на следующий элемент. Доступ к элементам контейнера осуществляется с помощью выражения ****i***.

Пользователям нет необходимости знать точный тип итераторов стандартных контейнеров. Тип итератора определяется в рамках определения контейнера и на него можно ссылаться по имени. Когда не нужно модифицировать элементы контейнера, целесообразно использовать итераторы типа ***const_iterator***. В противном случае нужно применять итераторы типа ***iterator*** (§16.3.1).

Добавлять элементы к списку и удалять их из него очень просто:

```
void f(const Entry& e, list<Entry>::iterator i, list<Entry>::iterator p)
{
  phone_book.push_front(e);   // добавить в начало
  phone_book.push_back(e);    // добавить в конец
  phone_book.insert(i,e);     // добавить перед элементом, указуемым через i

  Phone_book.erase(p);        // удалить элемент, указуемый через p
}
```

Детальное описание функций ***insert***() и ***erase***() дано в §16.3.6.

3.7.4. Контейнер map

Писать программы для поиска имен в списках пар значений (имя, номер телефона) довольно утомительно. Кроме того, последовательный поиск эффективен лишь для коротких списков. В то же время, существуют специальные структуры данных, которые напрямую поддерживают внедрение элементов, их удаление и поиск по значению. В стандартной библиотеке, например, имеется контейнерный тип

map (§17.4.1). Тип ***map*** является контейнером для хранения пар величин. Например:

```
map<string, int> phone_book;
```

Тип ***map*** часто называют *ассоциативным массивом* (*associative array*) или *словарем* (*dictionary*).

Будучи индексирован по значению его первого типа (называемого *ключом* — *key*), ***map*** возвращает соответствующее значение его второго типа (называемого просто *значением* или *отображаемым типом* — *value* или *mapped type*). Например:

```
void print_entry(const string& s)
{
  if(int i=phone_book[s]) cout<<s<<' '<<i<<'\n';
}
```

Если вхождение ключа *s* не обнаружено, возвращается некоторое значение «по умолчанию». Для целочисленных типов данных в контейнерах типа ***map*** таковым служит *нуль*. А в нашем случае *нуль* не является допустимым телефонным номером.

3.7.5. Контейнеры стандартной библиотеки

Контейнеры типов ***map***, ***list*** и ***vector*** могут использоваться для представления телефонной книги. У каждого из них есть свои *сильные и слабые стороны*. Например, индексирование векторов является простой и эффективной операцией. Но с другой стороны, внедрение элемента между существующими элементами *вектора* — весьма затратная операция. Для *списков* ситуация во всем противоположная. Ассоциативный массив (***map***) подобен *списку пар* (ключ, значение), но он оптимизирован под поиск значений по заданному ключу.

В стандартной библиотеке реализовано множество полезных типов контейнеров, из которых программист всегда может выбрать тот, который больше всего подходит к особенностям конкретной задачи:

Стандартные контейнеры	
vector<T>	вектор переменного размера (§16.3)
list<T>	двусвязный список (§17.2.2)
queue<T>	очередь (§17.3.2.)
stack<T>	стек (§17.3.1.)
deque<T>	двусторонняя очередь (§17.2.3.)
priority_queue<T>	очередь с приоритетом (§17.3.3.)
set<T>	множество (§17.4.3.)
multiset<T>	множество, в котором ключи могут встречаться несколько раз (§17.4.4.)
map<key, val>	ассоциативный массив (§17.4.1.)
multimap<key, val>	ассоциативный массив, в котором ключ (key) может встречаться несколько раз (§17.4.2.)

Подробнее стандартные контейнеры рассматриваются в §16.2, §16.3 и главе 17. Они объявляются в пространстве имен ***std*** и реализуются в заголовочных файлах **<*vector*>**, **<*list*>**, **<*map*>** и так далее (§16.2).

Стандартные контейнеры и их базовые операции разрабатывались так, чтобы с понятийной точки зрения они были очевидными. Более того, одни и те же операции для разных контейнеров имеют один и тот же смысл. В общем случае, базовые операции применимы ко всем типам контейнеров. Например, операция ***push_back***() может использоваться (с разумной эффективностью) для добавления элементов в конец *векторов* и *списков*, а операция ***size***() сообщает количество элементов для любого контейнера.

Такая *выразительная и семантическая однородность* помогает программистам создавать новые типы контейнеров, которые можно использовать так же, как и стандартные типы контейнеров. Примером может служить рассмотренный выше тип ***Vec*** (§3.7.2), проверяющий допустимость значений индексов. В главе 17 будет рассмотрен контейнерный тип ***hash_map***. Однородность контейнерных интерфейсов *позволяет разрабатывать алгоритмы независимо от конкретных типов контейнеров.*

3.8. Алгоритмы

Структуры данных, вроде списка или вектора, сами по себе не очень-то и полезны. Чтобы их можно было использовать, требуются такие базовые операции, как добавление и удаление элементов. Более того, мы редко используем контейнеры исключительно с целью хранения объектов. Чаще всего, мы объекты сортируем, печатаем, удаляем, извлекаем их подмножества, выполняем поиск и т.д. Как следствие, в стандартной библиотеке наряду с общеупотребительными контейнерами содержатся и *общеупотребительные алгоритмы*. Например, определив операции == (равно) и < (меньше) для типа ***Entry***, мы можем отсортировать вектор ***vector<Entry>*** и положить копии всех его уникальных элементов в список:

```
bool operator==(const Entry& a, const Entry& b) { return a.name==b.name; }
bool operator<(const Entry& a, const Entry& b) { return a.name<b.name; }

void f(vector<Entry>& ve, list<Entry>& le)
{
  sort( ve.begin(),ve.end() );
  unique_copy( ve.begin(),ve.end(),le.begin() );
}
```

Стандартные алгоритмы рассматриваются в главе 18. Они формулируются в терминах *последовательностей элементов* (*sequences of elements*) (§2.7.2). Последовательность специфицируется *парой итераторов*, указывающих на ее первый элемент, и на «элемент, следующий за последним». В примере функция ***sort***() сортирует последовательность элементов, начиная с ***ve.begin***() и заканчивая ***ve.end***() — здесь это соответствует всем элементам вектора ***ve***. Для операции записи достаточно указать лишь первый из элементов, подлежащих перезаписи. Если в операции участвует не один элемент, то будут перезаписаны все элементы целевого объекта, следующие за указанным.

Можно также добавить новые элементы в конец контейнера:

```
void f(vector<Entry>& ve, list<Entry>& le)
{
  sort(ve.begin(),ve.end());
  unique_copy(ve.begin(),ve.end(),back_inserter(le));     // добавить в конец le
}
```

С помощью ***back_inserter***() элементы добавляются в *конец* контейнера, при этом емкость контейнера автоматически расширяется до необходимых размеров (§19.2.4). Таким образом, стандартные контейнеры и ***back_inserter***() позволяют полностью избавиться от потенциально опасного управления памятью с помощью библиотечной функции ***realloc***() языка С (§16.3.5).

Отметим, что если вы забудете про ***back_inserter***() при добавлении элементов, то придете к разного рода ошибочным ситуациям. Например:

```
void f(vector<Entry>& ve, list<Entry>& le)
{
  copy(ve.begin(),ve.end(),le);                 // ошибка: le не итератор
  copy(ve.begin(),ve.end(),le.end());           // очень плохо: просто пишем за конец
  copy(ve.begin(),ve.end(),le.begin());         // перезаписывает элементы
}
```

3.8.1. Использование итераторов

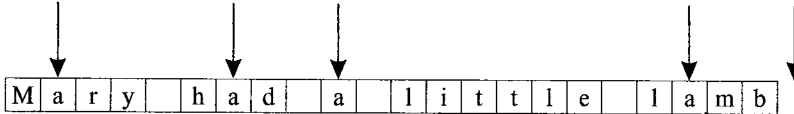
Располагая контейнером, легко получить итераторы, указывающие на характерные элементы контейнера, например, вызвав функции ***begin***() и ***end***(). Кроме того, многие алгоритмы возвращают итераторы. Например, *стандартный алгоритм* ***find***() ищет значение в последовательности и возвращает итератор, указывающий на найденный элемент. Используя ***find***(), можно найти число вхождений символа в строку:

```
int count(const string& s, char c)
{
  int n = 0;
  string::const_iterator i = find(s.begin(),s.end(),c);

  while( i != s.end() )
  {
    ++n;
    i = find(i+1,s.end(),c);
  }
  return n;
}
```

Функция ***find***() возвращает либо итератор, указывающий на найденный элемент, либо итератор, указывающий на «элемент, следующий за последним». Рассмотрим, что происходит, когда вызывается функция ***count***():

```
void f()
{
  string m = "Mary had a little lamb";
  int a_count = count(m,'a');
}
```

Первый вызов алгоритма ***find***() найдет символ *'a'* в слове ***Mary***. В результате мы входим в цикл, так как возвращаемый при этом итератор указывает на внутренний символ строки и, естественно, не равен ***s.end***(). В цикле мы вызываем ***find***(), используя ***i+1*** для начального итератора последовательности, то есть продолжаем поиск с символа, следующего за ранее найденным символом *'a'*. В результате циклических вызовов алгоритма ***find***() мы находим три остальных символа *'a'*. В конце концов, ***find***() возвратит значение, равное ***s.end***(), так что условие ***i != s.end***() станет ложным и цикл завершится. Все это удобно проиллюстрировать рисунком:



Здесь стрелки отображают начальное, промежуточные и конечное значения итератора ***i***.

Естественно, алгоритм ***find***() будет работать аналогично с любым другим стандартным контейнером. Поэтому мы могли бы обобщить функцию ***count***() следующим образом:

```
template<class C, class T> int count(const C& v, T val)
{
  typename C::const_iterator i=find(v.begin(),v.end(),val); // typename см. §C.13.5
  int n = 0;

  while(i!= v.end())
  {
    ++n;
    ++i;        // пропустить только что найденный элемент
    i=find(i,v.end(),val);
  }
  return n;
}
```

Вот пример, использующий новый вариант функции ***count***() (*тип* ***complex*** определен в файле **<complex>**; §3.9.1, §22.5):

```
void f(list<complex<double> >& lc, vector<string>& vs, string s)
{
  int i1 = count(lc,complex<double>(1,3));
  int i2 = count(vs,"Chrysippus");
  int i3 = count(s,'x');
}
```

На самом деле, нам нет нужды определять собственный функциональный шаблон ***count***(). Задача подсчета числа вхождений элемента настолько важна, что стандартная библиотека предоставляет соответствующий алгоритм. Для большей универсальности *алгоритм* ***count***() из стандартной библиотеки принимает последовательность в качестве аргумента, а не контейнер. Так что клиентскую функцию ***f***() следует переписать в следующем виде:

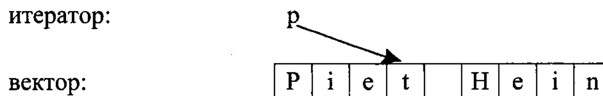
```
void f(list<complex<double> >& lc, vector<string>& vs, string s)
{
  int i1=count(lc.begin(),lc.end(),complex<double>(1,3));
  int i2=count(vs.begin(),vs.end(),"Diogenes");
  int i3=count(s.begin(),s.end(),'x');
}
```

Стандартный алгоритм ***count***() *может работать* и *со встроенными массивами*, и с *отдельными частями контейнеров*. Например:

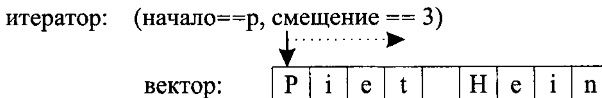
```
void g(char cs[], int sz)
{
  int i1=count(&cs[0],&cs[sz],'z');    // число символов z в массиве
  int i2=count(&cs[0],&cs[sz/2],'z'); // число символов z в первой половине массива
}
```

3.8.2. Типы итераторов

Чем на самом деле являются итераторы? В общем случае, про них можно сказать, что это объекты некоторого типа. Фактически же, *типы у разных итераторов разные*, так как конкретный итератор должен хранить информацию, позволяющую ему работать с конкретным типом контейнера. Типы итераторов могут отличаться в той же степени, в какой различаются типы контейнеров и цели, для достижения которых итераторы предназначены. Например, итераторы для контейнера ***vector*** скорее всего есть просто встроенные указатели, так как их удобно использовать для ссылок на индивидуальные элементы вектора:

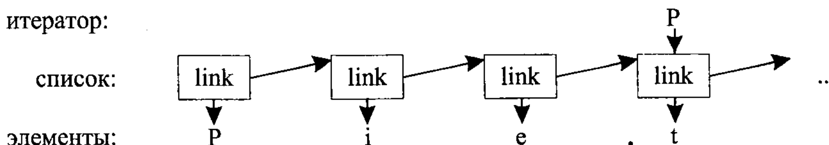


В качестве альтернативы можно реализовать итераторы в виде пары — указатель на начало ***вектора*** плюс целочисленное смещение:



Последнее решение позволяет реализовать проверку индексов (§19.3).

Итераторы для списков должны быть более сложными, чем просто встроенные указатели, так как элементы списков, в общем случае, не знают, где находится следующий элемент списка. Итератор для списка, гипотетически, мог бы быть указателем на связь (link) между элементами:



Общим для всех типов итераторов является их *семантика* и *обозначения операций*. Например, если к любому итератору применить операцию ++, то она вернет значение итератора, указывающего на следующий элемент. Аналогично, для любого итератора операция * возвращает элемент, на который настроен итератор. На самом деле, любой объект, подчиняющийся некоторым требованиям, вроде рассмотренной семантики операций, и будет фактически итератором (§19.2.1). Более того, пользователям в редких случаях нужно знать истинные типы итераторов; каждый контейнер сам «знает» типы своих итераторов и предоставляет их пользователям под стандартными именами ***iterator*** и ***const_iterator***. Например, для ***list<Entry>*** общим типом итераторов является тип ***list<Entry>::iterator***. Мне редко доводилось интересоваться деталями устройства этого типа.

3.8.3. Итераторы и ввод/вывод

Итераторы являются очень общей и полезной концепцией для манипулирования последовательностями элементов в контейнерах. Тем не менее, последовательности элементов встречаются *не только* в контейнерах. Например, *входной поток образует последовательность значений*; мы записываем последовательность значений в поток вывода. Как следствие, концепция итераторов естественным образом применяется ко вводу/выводу.

Для создания итераторного объекта типа ***ostream_iterator*** мы должны указать, какой поток будет использоваться и каков тип выводимых в него объектов. Например, мы можем определить *итератор, ссылающийся на стандартный поток вывода*, ***cout***:

```
ostream_iterator<string> oo (cout) ;
```

Если мы что-то присваиваем выражению ****oo***, то смыслом такого присваивания является запись присваиваемого значения в поток ***cout***. Например:

```
int main ()
{
  *oo ="Hello, ";          // означает cout <<?"Hello, "
  ++oo;
  *oo ="world!\n";         // означает cout <<?"world!\n"
}
```

Можно сказать, что это еще один способ осуществить стандартный вывод «канонического текста» ***Hello***, ***world***!. Выражение ***++oo*** имитирует запись в массив с помощью указателей. Конечно, я не стал бы на практике использовать показанный код для решения столь простой задачи, но заманчиво сразу же показать работу с потоками вывода как с *контейнерами «только для записи»* (*write-only container*); вскоре эта идея должна стать очевидной (если не уже стала очевидной).

Аналогично, ***istream_iterator*** есть нечто, что позволяет нам работать с потоком ввода как с *контейнером «только для чтения»* (*read-only container*). Мы должны указать используемый поток ввода и тип ожидаемых объектов:

```
istream_iterator<string> ii (cin) ;
```

Так как *входные итераторы* (*input iterators*) неизменно используются *парами*, специфицирующими последовательность элементов, мы должны еще предоставить

итератор, указывающий на конец ввода. Им является итератор типа ***istream_iterator***, создаваемый «конструктором по умолчанию»:

```
istream_iterator<string> eos;
```

Теперь мы можем ввести фразу ***Hello, world!***, а затем вывести ее:

```
int main()
{
  string s1=*ii;
  ++ii;
  string s2=*ii;

  cout<<s1<<' '<<s2<<'\n';
}
```

Вообще-то, итераторы типов ***ostream_iterator*** и ***istream_iterator*** не предназначены для непосредственного использования. Они, как правило, используются в качестве параметров алгоритмов. Например, следующая простая программа читает файл, сортирует прочитанные слова, устраняет дубликаты и выводит результат в другой файл:

```
int main()
{
  string from, to;
  cin>> from>> to;                        // получить имена исходного и целевого файлов

  ifstream is(from.c_str());              // поток ввода (c_str(); см §3.5.1 и §20.3.7)
  istream_iterator<string> ii(is);        // итератор ввода для потока
  istream_iterator<string> eos;           // страж ввода

  vector<string> b(ii,eos);               // b это вектор, инициализируемый вводом
  sort(b.begin(),b.end());                // сортируем буфер

  ofstream os(to.c_str());                // поток вывода
  ostream_iterator<string> oo(os,"\n");   // итератор вывода для потока

  unique_copy(b.begin(),b.end(),oo);      // копировать буфер в поток вывода,
                                          // удаляя повторяющиеся значения
  return !is.eof() || !os;                // возврат состояния ошибки (§3.2, §21.3.3)
}
```

Тип ***ifstream*** — это поток ввода ***istream***, который может быть *связан с файлом*, а ***ofstream*** — это поток вывода ***ostream***, который также может связываться с файлом. Второй аргумент конструктора, использованного для создания итератора ***oo*** типа ***ostream_iterator***, предназначен для *отделения друг от друга выводимых значений* (*delimit output values*).

3.8.4. Алгоритм for_each и предикаты

Для перебора всех элементов последовательностей приходится использовать итераторы в операторах циклах. Это может быть довольно утомительной задачей, так что стандартная библиотека предоставляет иные способы *вызова функций для каждого элемента последовательности.*

Рассмотрим программу для чтения слов из потока и вычисления частоты их встречаемости. Естественным типом данных, предназначенным для хранения слов вместе с их частотой, является ***map***:

```
map<string, int> histogram;
```

Для подсчета частоты встречаемости слов достаточно следующего очевидного кода:

```
void record(const string& s)
{
  histogram[s]++;           // частота вхождения s
}
```

Когда все данные введены и обработаны, нужно вывести результаты подсчетов. Контейнеры типа ***map*** содержат последовательности пар значений (string,int), имеющих *тип pair*, так что для вывода нужно будет вызвать функцию

```
void print(const pair<const string, int>& r)
{
  cout<<r.first<<' '<<r.second<<'\n';
}
```

для каждого элемента контейнера (первое поле типа ***pair*** называется ***first***, а второе — ***second***). Первое поле в ***pair*** является скорее ***const string***, чем просто ***string***, так как все ключи в контейнерах ***map*** есть константы.

Вот клиентский код для выполнения намеченной работы:

```
int main()
{
  istream_iterator<string> ii(cin);
  istream_iterator<string> eos;

  for_each(ii, eos, record);
  for_each(histogram.begin(), histogram.end(), print);
}
```

Нет необходимости сортировать контейнер ***map*** для получения упорядоченного вывода. Ассоциативные массивы *самостоятельно сортируют свои элементы*, так что в нашей программе перебор элементов автоматически осуществляется в возрастающем порядке.

Во многих задачах требуется найти что-либо в контейнере, а не просто выполнить заданное действие для каждого его элемента. Например, алгоритм ***find***() (§18.5.2) позволяет отыскивать в контейнере специфическое значение. Более общим вариантом этой задачи является поиск элемента, удовлетворяющего заданному требованию. Например, нам может потребоваться разыскать элемент, значение которого больше 42. Так как контейнеры типа map предоставляют свои элементы в виде пар значений (***key,value***), то нам нужно найти среди последовательности элементов контейнера ***map<string, int>*** такую пару ***pair<const string, int>***, для которой целочисленное поле больше 42:

```
bool gt_42(const pair<const string, int>& r)
{
  return r.second>42;
}

void f(map<string, int>& m)
{
```

```
    typedef map<string,int>::const_iterator MI;
    MI=find_if(m.begin(),m.end(),gt_42);
    // ...
}
```

Кроме того, мы могли бы подсчитать число слов с частотой встречаемости, большей 42:

```
void g(const map<string,int>& m)
{
    int c42 = count_if(m.begin(),m.end(),gt_42);
    // ...
}
```

Функции типа `gt_42()` используются для управления алгоритмами и называются *предикатами*. Предикат *вызывается для каждого элемента* и *возвращает логическое значение*, которое алгоритм использует для принятия решения — выполнять или не выполнять заданное действие над элементом. Например, *алгоритм* `find_if()` работает до тех пор, пока его предикат не вернет значение `true`, означающее, что искомый элемент найден. Аналогичным образом, *алгоритм* `count_if()` подсчитывает число элементов контейнера, для которых предикат возвращает значение `true`.

В стандартной библиотеке имеется ряд готовых предикатов и несколько полезных шаблонов, позволяющих получать новые предикаты (§18.4.2).

3.8.5. Алгоритмы, использующие функции-члены классов

Многие алгоритмы вызывают функции для каждого элемента последовательности. Например, в §3.8.4 алгоритм

```
for_each(ii,eos,record);
```

вызывает функцию `record()` для каждой строки, прочитанной из потока ввода.

Часто приходится работать с контейнерами, содержащими указатели на объекты, для которых по этим указателям нужно вызвать функции-члены классов, а не глобальные функции. Например, нам может потребоваться вызвать функцию-член `Shape::draw()` для каждого элемента контейнера типа `list<Shape*>`. Для решения данной конкретной задачи мы можем просто написать глобальную функцию, вызывающую функцию-член класса:

```
void draw(Shape* p)
{
    p->draw();
}

void f(list<Shape*>& sh)
{
    for_each(sh.begin(),sh.end(),draw);
}
```

Обобщением этой идеи является решение, опирающееся на шаблон `mem_fun()` из стандартной библиотеки:

```
void g (list<Shape*>& sh)
{
  for_each (sh.begin (), sh.end (), mem_fun (&Shape::draw));
}
```

Шаблон ***mem_fun***() (§18.4.4.2) в качестве параметра принимает указатель на функцию-член класса (§15.5) и возвращает нечто, что может быть вызвано по указателю на класс. Результат ***mem_fun***(***&Shape::draw***) берет аргумент типа ***Shape****** и возвращает то, что возвращает функция ***Shape::draw***().

Шаблон ***mem_fun*** очень важен на практике, так как позволяет использовать стандартные алгоритмы для контейнеров, содержащих объекты полиморфных типов.

3.8.6. Алгоритмы стандартной библиотеки

Что такое алгоритм? Общее определение алгоритма гласит, что это «конечный набор правил, определяющих последовательность операций для решения конкретного множества задач и удовлетворяющих пяти принципам: конечность ... определенность ... ввод ... вывод ... эффективность» [Knuth, 1968, §1.1]. В контексте стандартной библиотеки *алгоритм есть набор шаблонов для работы с последовательностями элементов.*

Стандартная библиотека содержит десятки алгоритмов. Алгоритмы объявляются в пространстве имен ***std*** и реализуются в заголовочном файле <***algorithm***>. Вот некоторые из наиболее полезных алгоритмов:

Нет перевода заголовка	
for_each()	Вызвать функцию для каждого элемента (§18.5.1)
find()	Найти первое вхождение аргументов (§18.5.2)
find_if()	Найти первое соответствие предикату (§18.5.2)
count()	Сосчитать число вхождений элемента (§18.5.3)
count_if()	Сосчитать число соответствий предикату (§18.5.3)
replace()	Заменить элемент новым значением (§18.6.4)
reptace_if()	Заменить элемент, соответствующий предикату, новым значением (§18.6.4)
copy()	Скопировать элементы (§18.6.1)
unique_copy()	Скопировать только различные элементы (§18.6.1)
sort()	Отсортировать элементы (§18.7.1)
equal_range()	Найти диапазон всех элементов с одинаковыми значениями (§18.7.2)
merge()	Слияние отсортированных последовательностей (§18.7.3)

Все эти и многие другие алгоритмы (см. главу 18) можно применять к стандартным контейнерам, строкам типа ***string*** и встроенным массивам.

3.9. Математические вычисления

Как и язык C, C++ не разрабатывался специально для решения вычислительных задач. И тем не менее, математическая проблематика реально присутствует в программах на языке C++, и стандартная библиотека этот факт отражает.

3.9.1. Комплексные числа

Стандартная библиотека поддерживает семейство комплекснозначных типов по линии класса ***complex***, рассмотренного в §2.5.2. Чтобы предоставить возможность скалярным элементам комплексных чисел иметь типы ***float***, ***double*** и так далее, класс ***complex*** стандартной библиотеки объявлен *шаблоном*:

```
template<class scalar> class complex
{
public:
  complex(scalar re, scalar im);
  // ...
};
```

Для комплексных чисел поддерживаются обычные арифметические операции и наиболее общие математические функции. Например:

```
// Стандартная функция возведения в степень из <complex>:
template<class C> complex<C> pow(const complex<C>&, int);

void f(complex<float> fl, complex<double> db)
{
  complex<long double> ld = fl+sqrt(db);
  db += fl*3;
  fl=pow(1/fl,2);
  // ...
}
```

Подробности см. в §22.5.

3.9.2. Векторная арифметика

Класс ***vector***, рассмотренный в §3.7.1, разрабатывался так, чтобы обеспечить общий и гибкий механизм для хранения объектов, который согласуется с архитектурой контейнеров, итераторов и алгоритмов. Этот класс не поддерживает математических операций с векторами. Добавить в класс ***vector*** такие операции довольно просто, но его общность и гибкость препятствуют достижению высокой эффективности, которой придается важное значение в серьезных вычислительных задачах. Как следствие, в стандартной библиотеке имеется еще один *векторный тип*, который называется ***valarray***. Он не настолько общий, зато хорошо поддается оптимизации при вычислениях:

```
template<class T> class valarray
{
  // ...
  T& operator[](size_t);
  // ...
};
```

Тип **size_t** есть беззнаковый целочисленный тип, который используется в конкретных системах для индексации массивов.

Для класса **valarray** реализованы обычные арифметические операции и наиболее общие математические функции. Например:

```
// Стандартная функция из <valarray>, вычисляющая модуль:
template<class T> valarray<T> abs(const valarray<T>&);

void f(valarray<double>& a1, valarray<double>& a2)
{
  valarray<double> a = a1*3.14+a2/a1;
  a2 += a1*3.14;
  a = abs(a);
  double d = a2[7];
  //...
}
```

Подробности см. в §22.4.

3.9.3. Поддержка базовых вычислительных операций

Естественно, стандартная библиотека содержит элементарные математические функции — такие как **log**(), **pow**() и **cos**() для типов с плавающей запятой; см. §22.3. Для всех встроенных типов предусмотрены классы, описывающие их характерные свойства, например, наибольший показатель степени для чисел с плавающей запятой; см. §22.2.

3.10. Основные средства стандартной библиотеки

Средства стандартной библиотеки можно классифицировать следующим образом:

1. Базовая поддержка языка программирования (выделение памяти; определение типа на стадии выполнения); см. §16.1.3.
2. Стандартная библиотека языка С (с минимальнейшими изменениями для предотвращения нарушений в системе типов); см. §16.1.2.
3. Строки и потоки ввода/вывода (с поддержкой национальных алфавитов и локализации); см. главы 20 и 21.
4. Набор контейнеров (таких как **vector**, **list** и **map**) и алгоритмов для работы с ними (прохода по всем элементам, сортировки, слияния и т.д.); см. главы 16, 17, 18 и 19.
5. Поддержка численных расчетов (комплексные числа, вектора для математических расчетов, BLAS-подобные и обобщенные срезы, семантика оптимизации); см. главу 22.

Главными критериями для включения класса в стандартную библиотеку являлись следующие соображения: будет ли он использоваться любыми С++-программистами (экспертами и новичками); может ли он быть представлен в общей форме, не влекущей за собой заметных накладных расходов по сравнению с более простыми средствами; можно ли легко научиться его простейшему применению. По сути

дела, *стандартная библиотека* предоставляет наиболее *фундаментальные реализации основных структур данных* вкупе с *фундаментальными алгоритмами их обработки.*

Любой алгоритм работает с любым контейнером без каких-либо преобразований. Такая структура библиотеки, обычно называемая STL (Standard Template Library; Stepanov, 1994), позволяет пользователям легко создавать свои собственные контейнеры и алгоритмы помимо имеющихся стандартных, и заставлять их согласованно работать с последними.

3.11. Советы

1. Не изобретайте колесо заново; используйте библиотеки.
2. Не верьте в чудеса; разберитесь, что делают ваши библиотеки, как они это делают и какова цена их деятельности.
3. Когда есть выбор, отдавайте предпочтение стандартной библиотеке.
4. Не думайте, что стандартная библиотека идеально подходит для всех задач.
5. Не забывайте про директиву *#include* для включения заголовочных файлов с определением средств стандартной библиотеки; §3.3.
6. Помните, что все средства стандартной библиотеки определены в пространстве имен *std*; §3.3.
7. Вместо строк типа *char** пользуйтесь строками типа *string*; §3.5, §3.6.
8. Если сомневаетесь — используйте вектора с проверкой диапазона индексов (такие как *Vec*); §3.7.2.
9. Используйте *vector<T>*, *list<T>*, *map<key,value>* вместо `T[]`; §3.7.1, §3.7.3, §3.7.4.
10. При добавлении элементов в контейнер используйте *push_back*`()` или *back_inserter*`()`; §3.7.3, §3.8.
11. Лучше использовать *push_back*`()` для векторов, чем *realloc*`()` для встроенных массивов; §3.8.
12. Самые общие исключения перехватывайте в стартовой функции *main*`()`; §3.7.2.

Часть I

Основные средства

Здесь описываются встроенные типы языка C++ и средства построения программ на их основе. Представлено C-подмножество языка C++ и его дополнительные возможности для поддержки традиционных стилей программирования. Также обсуждаются базовые средства составления программ на C++ из логических и физических частей.

Главы

4. Типы и объявления

5. Указатели, массивы и структуры

6. Выражения и операторы

7. Функции

8. Пространства имен и исключения

9. Исходные файлы и программы

4

Типы и объявления

Не соглашайтесь ни на что, кроме совершенства!
— Аноним

Совершенство достигается только к моменту краха.
— К.Н. Паркинсон

Типы — фундаментальные типы — логические значения — символы — символьные литералы — целые — целые литералы — типы с плавающей запятой — литералы с плавающей запятой — размеры — ***void*** — перечисления — объявления — имена — область видимости — инициализация — объекты — ***typedef***— советы — упражнения.

4.1. Типы

Рассмотрим формулу

```
x = y + f(2) ;
```

Чтобы это выражение могло использоваться в программе на С++, имена ***x***, ***y*** и ***f*** должны быть *объявлены* надлежащим образом. То есть программист должен указать, что некие сущности, поименованные ***x***, ***y*** и ***f***, реально существуют, и что они имеют типы, для которых операции = (присваивание), + (сложение) и () (вызов функции) имеют точный смысл.

Каждое *имя* (*идентификатор*) в программе на С++ *ассоциируется с некоторым типом.* Именно тип определяет, какие операции можно выполнять с этим именем (то есть с именованной сущностью), и как интерпретируются эти операции. Например, следующие объявления

```
float x;          // x - переменная с плавающей запятой
int y=7;          // y - целая переменная с начальным значением 7
float f(int) ;    // f - функция с аргументом целого типа,
                  // возвращающая число с плавающей запятой
```

делают наш текущий пример осмысленным. Поскольку y объявлена переменной целого типа, то ей можно присваивать значения, ее можно использовать в арифметических выражениях и т.д. С другой стороны, *f* объявлена как функция, принимающая целый аргумент, так что ее можно вызывать с подходящим аргументом.

В данной главе рассматриваются фундаментальные типы (§4.1.1) и объявления (§4.9). Примеры в ней просто иллюстрируют свойства языка; они не предназначены для демонстрации чего-либо практически полезного. Изощренные и реалистичные примеры будут приводиться в последующих главах, когда большая часть языка C++ будет уже рассмотрена. Здесь же перечисляются базовые элементы, из которых состоят все программы на языке C++. Вы должны освоить эти элементы плюс терминологию и сопутствующий им несложный синтаксис, чтобы выполнять законченные проекты на C++, и особенно для того, чтобы иметь возможность разбираться в чужих кодах. Тем не менее, для изучения последующих глав глубокого понимания всех рассмотренных в настоящей главе вопросов не требуется. Достаточно бегло рассмотреть основные концепции, а за необходимыми деталями вернуться позднее.

4.1.1. Фундаментальные типы

Язык C++ обладает набором *фундаментальных типов*, соответствующих *базовым принципам организации компьютерной памяти* и самым общим способам хранения данных:

- §4.2 Логический тип (***bool***)
- §4.3 Символьные типы (такие как ***char***)
- §4.4 Целые типы (такие как ***int***)
- §4.5 Типы с плавающей запятой (такие как ***double***)

Дополнительно, пользователь может объявить

- §4.8 Перечислимые типы для представления конечного набора значений (***enum***)

Также имеется

- §4.7 Тип ***void***, указывающий на отсутствие информации

Поверх перечисленных типов можно строить следующие типы:

- §5.1 Указательные типы (такие как ***int****)
- §5.2 Массивы (такие как ***char***[])
- §5.5 Ссылочные типы (такие как ***double***&)
- §5.7 Структуры данных и классы (глава 10)

Логический, символьные и целые типы называются *интегральными типами*. Интегральные типы и типы с плавающей запятой в совокупности называются *арифметическими типами*. Перечисления и классы (глава 10) называются *типами, определяемыми пользователем*, так как их нужно определить до момента использования, что контрастирует с фундаментальными типами, не требующими никаких объявлений. Поэтому их еще называют *встроенными типами* (*built-in types*).

Интегральные типы и типы с плавающей запятой *могут иметь различные размеры*, предоставляя тем самым программисту выбор объема занимаемой ими памяти, точности представления и диапазона возможных значений (§4.6). Общая идея со-

стоит в том, что на компьютере символы хранятся в виде набора байт, целые числа хранятся и обрабатываются как наборы машинных слов, числа с плавающей запятой располагаются в характерных для них объемах памяти (соответствующих специализированным регистрам процессора), и на все эти сущности можно ссылаться по адресам. Фундаментальные типы языка C++ вместе с указателями и массивами предоставляют программисту отражение этого машинного уровня в манере, относительно независимой от конкретной реализации.

В большинстве приложений можно обойтись типом ***bool*** для логики, типом ***char*** для символов, типом ***int*** для целых чисел и типом ***double*** — для дробных (чисел с плавающей запятой). Остальные фундаментальные типы являются вариациями, предназначенными для оптимизации и решения специальных задач, и их лучше до поры игнорировать. Но, разумеется, их нужно знать для чтения существующего стороннего кода на C и C++.

4.2. Логический тип

Переменные логического типа ***bool*** могут принимать лишь два значения — ***true*** (истина) или ***false*** (ложь). Логические переменные служат для хранения результатов логических операций. Например:

```
void f(int a, int b)
{
  bool bl = a==b;   // = есть присваивание, == проверка на равенство;
  // ...
}
```

Если у целых переменных ***a*** и ***b*** значения одинаковые, то ***b1*** будет равно ***true***; в противном случае ***b1*** станет равным ***false***.

Часто тип ***bool*** служит типом возврата *функций, проверяющих выполнение некоторого условия* (*предикатные функции*, или просто *предикаты*). Например:

```
bool is_open(File*);
bool greater(int a, int b) {return a>b;}
```

По определению, при преобразовании к типу ***int*** значению ***true*** сопоставляется ***1***, а значению ***false*** — ***0***. В обратном направлении целые значения неявно преобразуются в логические следующим образом: ненулевые значения трактуются как ***true***, а ***0*** как ***false***. Например:

```
bool b=7;        // bool(7) есть true, так что b инициализируется значением true
int i= true;     // int(true) есть 1, так что i инициализируется значением 1
```

В арифметических и логических выражениях логические значения сначала преобразуются в целые, после чего выражения и вычисляются. Если результат вычислений требуется преобразовать обратно в тип ***bool***, то ***0*** преобразуется в ***false***, а ненулевые значения — в ***true***:

```
void g()
{
  bool a=true;
  bool b=true;
```

```
  bool x=a+b;     // a+b равно 2, так что x становится true
  bool y=a|b;     // a|b равно 1, так что y становится true
}
```

Указатель можно неявно преобразовывать к типу ***bool*** (§C.6.2.5). Ненулевой указатель преобразуется в ***true***; указатель с нулевым значением — в ***false***.

4.3. Символьные типы

В переменной типа ***char*** может храниться код символа, соответствующий некоторой стандартной кодировке. Например:

```
char ch='a';
```

Практически повсеместно под тип ***char*** отводится *8 бит памяти*, так что переменные этого типа могут хранить *один из 256 символов*. В типичном случае кодировка символов является одним из вариантов стандарта ISO-646, например ASCII, что соответствует символам стандартной клавиатуры. Много проблем проистекает из-за того, что набор этот стандартизован не полностью (§C.3).

Серьезные различия имеют место для кодировок, поддерживающих разные национальные языки, и даже для разных кодировок одного и того же языка. Здесь мы, однако, интересуемся лишь тем, как это влияет на язык C++. Сложная и интересная проблема создания программ, поддерживающих несколько национальных языков и/или кодировок, в целом выходит за рамки настоящей книги, хотя и упоминается в отдельных ее местах (§20.2, §21.7 и §C.3.3).

Можно с уверенностью полагать, что любая кодировка содержит десятичные цифры, 26 букв английского алфавита и стандартные знаки пунктуации. Неправильно полагать, что каждая 8-битная кодировка содержит не более 127 символов (большинство национальных кодировок содержат 255 символов), что нет алфавитных символов, отличных от букв английского языка (многие Европейские языки содержат дополнительные символы), что алфавитные символы имеют смежные коды (кодировка EBCDIC оставляет зазор между символами '*i*' и '*j*'), или что в обязательном порядке присутствуют все символы, необходимые для записи конструкций языка C++ (в некоторых национальных кодировках отсутствуют символы { } [ ] | \; см. §C.3.1). Лучше вообще избегать привязки к конкретным представлениям объектов. Это касается и символов.

Каждому символу сопоставляется целочисленное значение. Например, символу '*b*' в рамках кодировки ASCII соответствует число 98. Вот программа, которая ответит на вопрос о числовом коде любого символа, который вы введете с клавиатуры:

```
#include <iostream>

int main()
{
  char c;
  std::cin>>c;
  std::cout<<"the value of '"<< c <<"' is "<<int(c)<<'\n';
}
```

Выражение *int(c)* дает целочисленное значение для символа *c*. В связи с преобразованием ***char*** в ***int*** возникает вопрос: тип char знаковый или беззнаковый? Ведь в зависимости от этого 256 восьмибитных значений могут трактоваться как числа от 0 до 255, или как числа от −128 до +127. К сожалению, точный ответ на вопрос *зависит от конкретной реализации* (§C.2, §C.3.4). В языке C++ два типа дают точные ответы на указанный вопрос: тип unsigned char соответствует значениям от 0 до 255, а тип ***signed char*** — значениям от −128 до +127. К счастью, различия касаются символов с кодами вне диапазона от 0 до 127, в который попадают все общеупотребительные символы.

Если тип ***char*** используется для хранения значений вне диапазона 0 — 127, то возможны проблемы с переносимостью. Ознакомьтесь с разделом §C.3.4, если у вас в программе используются сразу несколько символьных типов или если вы храните целые значения в типе ***char***.

Тип ***wchar_t*** специально предназначен для хранения более широкого диапазона кодов, характерных, например, для кодировки Unicode. Размер этого типа данных зависит от реализации и всегда достаточен для хранения самого широкого символьного набора, поддерживающего ту или иную национальную специфику (см. §21.7 и §C.3.3). Странное имя этого типа досталось от языка C. В языке C этот тип не является встроенным, а определяется с помощью оператора ***typedef*** (§4.9.7). Суффикс ***_t*** специально применяется с целью четкого указания на то, что *тип определен с помощью* ***typedef***.

Подчеркнем, что символьные типы являются интегральными (§4.1.1), так что к ним можно применять арифметические и побитовые логические операции (§6.2).

4.3.1. Символьные литералы

Символьным литералом (*символьной константой*; *character literal* или *character constant*) называется символ, заключенный в *апострофы* (одиночные кавычки), например, *'a'* или *'0'*. Типом символьного литерала является ***char***. Символьные литералы на самом деле есть просто символьные константы, именующие целочисленные значения из текущих кодировок символов, используемых на компьютере. Например, если программа исполняется на компьютере с ASCII-кодировкой, то *'0'* есть 48. Применение символьных литералов вместо десятичных обозначений делает программы более переносимыми. Ряд символьных литералов в своей записи используют специальный символ \ (backslash — обратная косая черта или escape-символ) и имеют стандартные названия, например, *'\n'* это символ перевода строки, а *'\t'* — символ табуляции. Более подробно такие символьные литералы рассматриваются в §C.3.2.

Символьные литералы для расширенных наборов символов имеют вид *L'ab'*, где количество символов между апострофами и их смысл зависят от конкретных реализаций, чтобы соответствовать типу ***wchar_t***. Естественно, что такие литералы имеют тип ***wchar_t***.

4.4. Целые типы

Как и тип ***char***, каждый целый тип представим в трех формах: «просто» ***int***, ***signed int*** и ***unsigned int***. Кроме того, целые могут быть трех размеров: ***short int***, «просто» ***int*** и ***long int***. Вместо ***long int*** можно просто писать ***long***. Аналогично, ***short*** есть синоним для ***short int***, ***unsigned*** — для ***unsigned int***, а ***signed*** — для ***signed int***.

Беззнаковые (*unsigned*) *целые типы* идеальны для трактовки блоков памяти как битовых массивов. Использование ***unsigned*** вместо ***int*** с целью заполучить лишний бит для представления целых положительных значений почти всегда является неудачным решением. А попытки гарантировать положительность числовых значений объявлением целой переменной с модификатором ***unsigned*** опровергаются правилами неявных преобразований типов (§С.6.1, §С.6.2.1).

В отличие от типа «просто» ***char***, *типы «просто»* ***int*** *всегда знаковые* (*signed*). Явное применение модификатора ***signed*** лишь подчеркивает этот факт.

4.4.1. Целые литералы

Целые литералы представимы в *четырех внешних обликах*: *десятичном*, *восьмеричном*, *шестнадцатеричном* и *символьном* (§А.3). Чаще всего используются десятичные литералы, которые выглядят вполне ожидаемым образом

0 1234 976 12345678901234567890

Компилятор обязан выдавать предупреждение, если величина литерала выходит за допустимые границы.

Литерал, начинающийся с ***0x***, является шестнадцатеричным (по основанию 16) литералом. Если же литерал начинается с нуля, но символ ***x*** за ним не следует, то это восьмеричный (по основанию 8) литерал. например:

decimal (***десятичный***) :		***2***	***63***	***83***
octal (***восьмеричный***) :	***0***	***02***	***077***	***0123***
hexadecimal (***шестнадцатеричный***) :	***0x0***	***0x2***	***0x3f***	***0x53***

Буквы ***a***, ***b***, ***c***, ***d***, ***e*** и ***f*** (или их эквиваленты в верхнем регистре) используются для обозначения чисел ***10***, ***11***, ***12***, ***13***, ***14*** и ***15***, соответственно. Восьмеричные и шестнадцатеричные литералы используются преимущественно для представления битовых последовательностей. Использование таких литералов для обозначения обычных чисел чревато сюрпризами. Например, на аппаратных платформах, у которых тип int реализуется 16-битным словом (с представлением отрицательных целых в дополнительном коде), ***0xffff*** есть отрицательное десятичное число ***−1***, а на платформах с большим числом бит — это уже ***65535***. Суффикс ***U*** явным образом обозначает *беззнаковое* (*unsigned*) целое, а суффикс ***L*** — *длинное* (*long*) целое. Например, типом литерала ***3*** является ***int***, типом ***3U*** — ***unsigned int***, а типом литерала ***3L*** — тип ***long int***. Если суффикс отсутствует, компилятор самостоятельно приписывает литералу подходящий тип, который он выбирает, сопоставляя значение литерала и машинозависимые размеры целых типов (§С.4).

Целесообразно ограничить использование неочевидных констант хорошо прокомментированными объявлениями с ключевым словом const (§5.4) или инициализаторами перечислений (§4.8).

4.5. Типы с плавающей запятой

Типы с плавающей запятой соответствуют числам с плавающей запятой. Как и целые типы, они могут быть *трех размеров*: ***float*** (одинарная точность), ***double*** (двойная точность) и ***long double*** (расширенная точность).

Точный смысл каждого типа зависит от реализации. Выбор оптимальной точности в конкретных задачах требует изрядных знаний в области вычислений с плавающей запятой. Если у вас их нет, то проконсультируйтесь со специалистом, или основательно изучите предмет, или используйте тип ***double*** наудачу.

4.5.1. Литералы с плавающей запятой

По умолчанию *литералы с плавающей запятой* (*floating-point literal*) имеют тип ***double***. Опять-таки, компилятор обязан выдавать предупреждения в случаях, когда значение литерала не соответствует машинным размерам для типов с плавающей запятой. Приведем примеры литералов с плавающей запятой:

1.23 .23 0.23 1. 1.0 1.2e10 1.23e-15

Пробельные символы в составе обозначений литералов с плавающей запятой *не допускаются*. Например, ***65.43 e-21*** не является литералом, а скорее это набор из четырех отдельных лексем (вызывающих ошибку компиляции):

65.43 e - 21

Если вам нужен литерал с плавающей запятой, имеющий тип ***float***, следует использовать *суффиксы **f*** или ***F***:

3.14159265f 2.0f 2.997925F 2.9e-3f

Для придания литералу типа long double следует использовать суффиксы ***l*** или ***L***:

3.14159265L 2.0L 2.997925L 2.9e-3L

4.6. Размеры

Некоторые аспекты фундаментальных типов языка C++, например, размер типа int, зависят от конкретных реализаций (§C.2). Я всегда обращаю внимание на подобного рода зависимости и рекомендую по возможности устранять их, или хотя бы минимизировать последствия. Почему вообще по этому поводу стоит беспокоиться? Людям, программирующим на разных платформах и применяющим разные компиляторы, ответ на вопрос очевиден: иначе придется тратить кучу времени, чтобы то и дело отыскивать трудноуловимые и крайне неочевидные ошибки. Люди же, ограниченные единственной системной платформой и единственным компилятором, чувствуют себя свободными от этой проблемы, утверждая, что «язык — это то, что реализует мой компилятор». Я считаю эту точку зрения узкой и недальновидной. Действительно, ведь если программа добивается успеха на какой-то платформе, то ее с огромной долей вероятности будут портировать на иные аппаратно-программные платформы, и кому-то все равно придется возиться с устранением имеющихся несовместимостей. Кроме того, часто программы приходится компилировать разными компиляторами на одной и той же системной платформе, или,

например, не исключена ситуация, когда новые версии одного и того же компилятора по-разному реализуют те или иные аспекты языка C++. Намного проще при первоначальном написании программного кода заранее выявить все источники системных несовместимостей и ограничить их влияние, чем потом разыскивать их в хитросплетениях готовой программы.

Ограничить влияние нюансов языка, зависящих от конкретной реализации, относительно несложно. Намного труднее ограничить влияние системнозависимых библиотек на переносимость программ. Везде, где только возможно, стоит использовать средства стандартных библиотек.

Несколько целых типов, беззнаковые типы и типы со знаком, нескольких типов с плавающей запятой — все это предназначено для того, чтобы программист мог *выжать максимум из аппаратных возможностей конкретного компьютера.* Для разных типов компьютеров объем требуемой памяти, время доступа к ней и скорость вычислений существенно зависят от выбора фундаментального типа. Если вам знакома машинная архитектура, то несложно выбрать, например, подходящий целый тип для конкретной переменной. Значительно труднее писать истинно переносимый низкоуровневый программный код.

Размеры программных объектов в C++ кратны размеру типа ***char***, так что фактически по определению размер типа ***char*** равен *1*. Размер любого объекта или типа можно узнать с помощью операции ***sizeof*** (§6.2). Вот что гарантируется относительно размеров фундаментальных типов:

$$1 \equiv sizeof(char) \le sizeof(short) \le izeof(int) \le sizeof(long)$$
$$1 \le sizeof(bool) \le sizeof(long)$$
$$sizeof(char) \le sizeof(wchar_t) \le sizeof(long)$$
$$sizeof(float) \le sizeof(double) \le sizeof(long\ double)$$
$$sizeof(N) \equiv sizeof(signed\ N) \equiv sizeof(unsigned\ N)$$

Здесь *N* может быть ***char***, ***short int***, ***int*** или ***long int***. Кроме того, гарантируется, что под ***char*** отводится минимум 8 бит, под ***short*** или ***int*** — минимум 16 бит, под ***long*** — минимум 32 бита. Тип ***char*** достаточен для представления всего набора символов, присущих машинной архитектуре.

Представим графически типовой набор фундаментальных типов, а также конкретный экземпляр текстовой строки:

char:	`'a'`
bool:	`1`
short:	`756`
int:	`100000000`
int*:	`&c1`
double:	`1234567e34`
char[14]:	`Hello, world!\0`

В сопоставимом масштабе (0.2 дюйма на байт) мегабайт памяти растянется вправо на три мили (около 5 километров).

Размер типа ***char*** выбирается так, чтобы он был оптимальным для хранения и манипулирования символами на данном типе компьютеров; обычно это 8 бит (один байт). Аналогично, размер типа ***int*** выбирается с целью оптимального хранения и вычислений с целыми числами; обычно это 4-байтовое слово (32 бита). Неразумно требовать большего, однако ж, машины с 32-битным типом char существуют.

Машинозависимые аспекты фундаментальных типов представлены в файле <***limits***> (§22.2). Например:

```
#include <limits>
#include <iostream>

int main ()
{
  std::cout<<"largest float=="<<std::numeric_limits<float>::max()
  << " , char is signed == " << std::numeric_limits<char>::is_signed<<'\n';
}
```

Разные фундаментальные типы можно свободно *смешивать в выражениях* (например, в присваиваниях). При этом, *по-возможности*, величины преобразуются так, чтобы *не терялась информация* (§C.6).

Если значение ***v*** может быть точно представлено в переменной типа ***T***, то преобразование ***v*** к типу ***T*** не ведет к потере информации, и нет проблем. Преобразований с потерей информации лучше всего просто не допускать (§C.6.2.6).

Для успешного выполнения больших программных проектов и для понимания практически важного стороннего кода требуется точное понимание процесса *неявного преобразования типов* (*implicit conversion*). Но для чтения последующих глав в этом нет необходимости.

4.7. Тип void

Синтаксически тип ***void*** относится к фундаментальным типам, но использовать его можно лишь как часть более сложных типов, ибо объектов типа ***void*** *не существует*. Этот тип используется либо для информирования о том, что у функции нет возврата, либо в качестве базового типа указателей на объекты неизвестных типов. Например:

```
void x;       // ошибка: не существует объектов типа void
void& r;      // ошибка: не существует ссылок на void
void f() ;    // функция f не возвращает значения (§7.3)
void* pv;     // указатель на объект неизвестного типа (§5.6)
```

Объявляя функцию, вы обязаны указать тип ее возврата. С логической точки зрения кажется, что в случае отсутствия возврата можно было бы просто опустить тип возврата. Это, однако, сделало бы синтаксис C++ менее последовательным (Приложение А) и конфликтовало бы с синтаксисом языка С. Поэтому ***void*** используется в качестве «псевдотипа возврата», означающего, что фактически возврата нет.

4.8. Перечисления

Перечисление (*enumeration*) является *типом*, содержащим *набор значений*, определяемых пользователем (программистом). После определения перечисление используется почти так же, как и целые типы.

Именованные целые константы служат *элементами перечислений*. Например, следующее объявление

```
enum {ASM, AUTO, BREAK};
```

определяет в качестве элементов перечисления три целые константы, которым неявно присваиваются целые значения. *По умолчанию*, целые значения присваиваются элементам перечисления в возрастающем порядке, *начиная с нуля*, так что ***ASM* == *0***, ***AUTO* == *1*** и ***BREAK* == *2***. Перечисление *может иметь имя*, например:

```
enum keyword { ASM, AUTO, BREAK};
```

Каждое перечисление является отдельным типом. *Типом элемента перечисления* является *тип самого перечисления*. Например, тип ***AUTO*** есть ***keyword***.

Если переменная объявляется с типом ***keyword*** (тип перечисления), а не просто int, то это дает пользователю и компилятору дополнительные сведения о предполагаемом характере использования этой переменной. Например:

```
void f(keyword key)
{
 switch (key)
 {
   case ASM:
     // некоторые действия
     break;
   case BREAK:
     // некоторые действия
     break;
 }
}
```

Здесь компилятор может выдать предупреждение о том, что используются лишь два из трех элементов перечисления.

Элементы перечисления можно явно инициализировать *константными выражениями* (§C.5) интегрального типа (§4.1.1). Когда все элементы перечисления неотрицательные, *диапазон их значений* равен $[0;2^k-1]$, где 2^k есть наименьшая степень двойки, для которой все элементы перечисления попадают в указанный диапазон. При наличии отрицательных элементов этот диапазон равен $[-2^k;2^k-1]$. Диапазон определяется *минимальным количеством бит, требуемых для представления всех значений элементов* (отрицательные значения — в дополнительном коде). Например:

```
enum e1 {dark, light};                  // диапазон 0:1
enum e2 {a = 3, b = 9};                 // диапазон 0:15
enum e3 {min = -10, max = 1000000};     // диапазон -1048576:1048575
```

Значение интегрального типа может быть явно преобразовано к типу перечисления. Если при этом исходное значение не попадает в диапазон перечисления, то результат преобразования не определен. Например:

```
enum flag {x=1, y=2, z=4, e=8 };        // диапазон 0:15

flag f1=5;                  // ошибка типа: 5 не принадлежит к типу flag
flag f2=flag(5);            // ok: flag(5) принадлежит типу flag и входит в его диапазон
flag f3=flag(z|e);          // ok: flag(12) принадлежит типу flag и входит в его диапазон
flag f4=flag(99);           // не определено: 99 не входит в диапазон типа flag
```

Последнее присваивание наглядно показывает, почему *не допускаются неявные преобразования целых в перечисления* — просто большинство целых значений не имеют представления в отдельно взятом конкретном перечислении.

Понятие диапазона перечисления в языке C++ *отличается от понятия перечисления в языках типа Pascal.* Однако в языке C и затем в языке C++ исторически закрепилась необходимость точного определения диапазона перечислений (не совпадающего со множеством значений элементов перечисления) для корректного манипулирования битами.

Под перечисления (точнее, под переменные этого типа) отводится столько же памяти (результат операции ***sizeof***), что и под интегральный тип, способный содержать весь диапазон перечисления. При этом указанный объем памяти не больше, чем ***sizeof(int)*** при условии, что все элементы перечисления представимы типами ***int*** или ***unsigned int***. Например, ***sizeof(e1)*** может быть равен ***1*** или ***4***, но не ***8***, на компьютере, для которого ***sizeof(int) == 4***.

При выполнении арифметических операций перечисления автоматически (неявно) преобразуются в целые (§6.2). *Перечисления относятся к определяемым пользователем типам*, так что для них *можно задавать свои собственные операции* вроде ++ или << (§11.2.3).

4.9. Объявления

Прежде, чем имя (идентификатор) будет использоваться в программе, оно должно быть объявлено. То есть должен быть специфицирован тип, говорящий компилятору о том, к какого рода сущностям относится поименованный программный объект. Вот примеры, иллюстрирующие разнообразие возможных объявлений:

```
char ch;
string s;
int count=1;
const double pi=3.1415926535897932385;
extern int error_number;
const char* name = "Njal";
const char* season[] = { "spring", "summer", "fall", "winter" };

struct Date{int d, m, y; };
int day(Date* p) {return p->d; }
double sqrt(double);
template<class T> T abs(T a) {return a<0 ? -a : a; }

typedef complex<short> Point;
struct User;
enum Beer {Carlsberg, Tuborg, Thor};
namespace NS { int a; }
```

Как видно из представленных примеров, объявления могут делать нечто большее, чем просто связывать имя с типом. Большинство из данных *объявлений* (*declarations*) являются еще и *определениями* (*definitions*), так как они помимо имени и типа определяют (создают) еще и сущности, на которые имена ссылаются. Для имени ch этой объективной сущностью является блок памяти определенного размера, выделяемый под переменную с именем ***ch***. Для имени day таковой сущностью является полностью определенная функция. Для константы ***pi*** — это число ***3.1415926535897932385***. Имя ***Date*** связывается с новым типом. Для ***Point*** объективной сущностью служит тип ***complex<short>*** — имя ***Point*** становится синонимом указанного типа. Из всех представленных выше объявлений только

```
double sqrt(double);
extern int error_number;
struct User;
```

не являются определениями: объявленные здесь имена ссылаются на сущности, которые должны быть определены в рамках иных объявлений. В некотором ином объявлении должно быть определено тело (код) функции ***sqrt***(), где-то в другом месте должна быть выделена память под целую переменную ***error_number***, в рамках иного объявления имени типа ***User*** должно быть точно специфицировано, что есть этот тип на самом деле, и где-то должен быть определен тип ***complex<short>***. Например:

```
double sqrt(double d) {/*...*/}
int error_number = 1;
struct User {/*...*/};
```

В программах на C++ *для каждой именованной сущности должно быть ровно одно определение* (о действии директивы ***#include*** см. §9.2.3). *Объявлений же может быть сколько угодно.* При этом *все объявления* некоторой сущности *должны согласовываться по типу* этой сущности. Поэтому фрагмент кода

```
int count;
int count;                  // ошибка: redefinition (повторное определение)
extern int error_number;
extern short error_number;  // ошибка: type mismatch (несоответствие типов)
```

содержит две ошибки, а следующий фрагмент

```
extern int error_number;
extern int error_number;

typedef complex<short> Point;
typedef complex<short> Point;
```

ошибок не содержит.

Некоторые определения снабжают определяемые сущности «значением». Например:

```
struct Date{int d, m, y;};
int day(Date* p) {return p->d;}
const double pi=3.1415926535897932385;
```

Для типов, шаблонов, функций и констант такое «значение» постоянно. Для неконстантных типов данных начальное значение может быть позднее изменено. Например:

```
void f()
{
  int count=1;
  const char* name="Bjarne"; // name указывает на константу (§5.4.1)
  // ...
  count=2;
  name="Marian";
}
```

Только в двух определениях из представленных в начале данного раздела не задаются значения:

```
char ch;
string s;
```

Вопрос о том, как и когда переменным присваиваются значения по умолчанию, рассмотрен в §4.9.5 и §10.4.2. Любое объявление, в котором значение задается, является определением.

4.9.1. Структура объявления

Объявление состоит из четырех частей: необязательного «спецификатора», базового типа, *декларатора* (*declarator*) и необязательного *инициализирующего выражения* (*инициализатора* — *initializer*). За исключением определений функции и пространства имен объявление заканчивается точкой с запятой. Например:

```
char* kings[] = {"Antigonus", "Seleucus", "Ptolemy"};
```

Здесь ***char*** является базовым типом, декларатором является **kings*[], а инициализирующее выражение есть ={...}. Стоящие в начале объявления ключевые слова, вроде ***virtual*** (§2.5.5, §12.2.6) или ***extern*** (§9.2), служат *спецификаторами* (*specifiers*), задающими (специфицирующими) *дополнительные атрибуты*, отличные от типа.

Деклараторы состоят из объявляемого имени и, возможно, дополнительных знаков операций. Наиболее употребительными в деклараторах являются следующие знаки операций (§A.7.1):

*	*указатель*	*prefix*
*const	*константный указатель*	*prefix*
&	*ссылка*	*prefix*
[]	*массив*	*postfix*
()	*функция*	*postfix*

Их использование не вызывало бы дополнительных сложностей, будь они все как один *префиксными* (*prefix*), или же *постфиксными* (*postfix*). Однако *, [] и () разрабатывались так, чтобы теснее отражать свою роль в построении выражений (§6.2). В результате, операция * является префиксной, а [] и () — постфиксные. В *деклараторах связь со знаками постфиксных операций более тесная, чем с префиксными*. Как следствие, **kings*[] есть массив указателей на что угодно, а в случае объявления «указателя на функцию» приходится в обязательном порядке использовать группирующие круглые скобки (см. примеры в §5.1). За более подробными грамматическими деталями отсылаем к Приложению А.

Отметим, что тип является неотъемлемой частью объявлений. Например:

```
const c=7;                           // error: нет типа
gt(int a, int b) {return (a>b) ? a : b; }  // error: нет типа возвращаемого значения
unsigned ui;                         // ok: 'unsigned' есть тип 'unsigned int'
long li;                             // ok: 'long' есть тип 'long int'
```

Ранние версии языков C и C++ отличались от текущего стандарта C++ тем, что первые два из перечисленных примеров являлись в них допустимыми, так как по умолчанию предполагался тип int (§B.2). Это *правило «неявного int»* служило *источником недоразумений* и приводило к трудноуловимым ошибкам.

4.9.2. Объявление нескольких имен

В одном операторе объявления можно сосредоточить сразу несколько объявляемых имен. В этом случае объявление содержит список деклараторов, разделенных запятыми. Например, можно объявить две целые переменные следующим образом:

```
int x, y;           // int x; int y;
```

Важно отметить, что *действие знаков операций относится исключительно к ближайшему имени*, и не распространяется на весь список имен в объявлении. Например:

```
int* p, y;          // int* p; int y; (не int* y;)
int x, *q;          // int x; int* q;
int v[10], *pv;     // int v[10]; int* pv;
```

В принципе, такие конструкции лишь затуманивают смысл программы, и их лучше избегать.

4.9.3. Имена

Имя (*идентификатор*) состоит из последовательности букв и цифр. Первым символом должна быть буква. Знак _ (знак подчеркивания) считается буквой. В языке C++ нет ограничений на количество символов в имени. Однако некоторые части системной реализации (например, компоновщик — linker) находятся вне компетенции разработчиков компилятора, и эти вот части, к сожалению, могут налагать определенные ограничения на имена. Среды выполнения (run-time environments) также могут расширять или ограничивать применимые в именах символы. Расширения (например, символ $) приводят к непереносимым программам. Ключевые слова языка C++ (Приложение A), например, new или int, не могут использоваться в качестве имен определяемых пользователем сущностей. Вот примеры допустимых имен:

```
hello      this_is_a_most_unusually_long_name
DEFINED    foO        bAr        u_name     HorseSense
var0       var1       CLASS      _class     ___
```

А вот примеры последовательностей символов, которые именами быть не могут:

```
012        a fool     $sys       lass 3var
pay.due    foo~bar    .name      if
```

Имена, начинающиеся со знака подчеркивания, обычно резервируются для специальных средств реализации и сред выполнения, так что их не стоит использовать в обычных прикладных программах.

Компилятор, читая программу, пытается для фиксации имен использовать *наиболее длинные последовательности символов*. Таким образом, последовательность символом ***var10*** составляет единое имя, а не имя ***var*** с последующим числом ***10***. Аналогично, ***elseif*** — это одно имя, а не ключевое слово ***else***, за которым следует ключевое слово ***if***.

Символы верхнего и нижнего регистров различаются, так что имена Count и count — разные. Однако создавать имена, отличающиеся лишь регистром их символов, не слишком разумно. И вообще, лучше избегать создания похожих друг на друга имен. Например, буква о в верхнем регистре (О) и нуль (0) выглядят похожими; это же характерно для буквы l и единицы (1). Следовательно, выбор идентификаторов l0, lO, l1 и ll вряд ли можно признать удачным.

Имена с широкой областью видимости должны быть достаточно длинными и «самоговорящими», например, ***vector***, ***Window_with_border*** или ***Department_number***. В то же время код становится чище и прозрачнее, если именам с узкой областью видимости присваиваются короткие и привычные имена вроде ***x***, ***i*** и ***p***. Классы (глава 10) и пространства имен (§8.2) помогают ограничивать области видимости. Полезно бывает часто используемые имена делать более короткими, а более длинные — резервировать для редко используемых сущностей. Целесообразно выбирать имена так, чтобы они отражали саму сущность, а не детали ее представления. Например, имя ***phone_book*** лучше, чем ***number_list***, несмотря на то, что телефонные номера и будут, вероятно, храниться в контейнере ***list*** (§3.7). Выбор хороших имен — это искусство.

Старайтесь придерживаться единого стиля именования. Например, имена пользовательских типов из нестандартной библиотеки начинайте с заглавной буквы, а имена, не относящиеся к типам — с прописной (например, ***Shape*** и ***current_token***). Имена макросов составляйте из одних лишь букв верхнего регистра (если уж нужен макрос, то пишите, например, HACK), а отдельные смысловые части идентификаторов разделяйте знаками подчеркивания. И все-таки, на практике единого стиля достигнуть трудно, так как программы часто составляются из фрагментов, полученных из разных источников, применяющих разумные, но разные системы именования. Будьте последовательны в выборе сокращений и составных имен.

4.9.4. Область видимости

Объявление вводит имя в *область видимости* (*scope*); это значит, что имя можно использовать лишь в ограниченной части программы. Для имени, объявленного в теле функции (его часто называют *локальным*), область видимости простирается от точки объявления до конца содержащего это объявление блока. *Блоком* называется фрагмент кода, ограниченного с двух сторон фигурными скобками (открывающей и закрывающей).

Имя называется *глобальным*, если оно объявлено вне функций, классов (глава 10) и пространств имен (§8.2). Область видимости глобального имени простирается от точки объявления до конца файла, содержащего это объявление. Объявление имени внутри блока скрывает совпадающее имя, объявленное во внешнем объемлющем блоке, или глобальное имя. Таким образом, внутри блока имя можно переопределить с тем, чтобы оно ссылалось на новую сущность. По выходе из блока имя восстанавливает свой прежний смысл. Например:

```
int x;              // глобальная переменная x

void f()
{
  int x;            // локальная x скрывает глобальную x
  x = 1;            // присваивание локальной x

  {
    int x;          // скрывает первую локальную x
    x = 2;          // присваивание второй локальной x
  }

  x = 3;            // присваивание первой локальной x
}

int* p = &x;        // взять адрес глобальной x
```

Сокрытие имен неизбежно в больших программах. В то же время, программист может и не заметить, что какое-то имя оказалось скрыто. Из-за того, что такие ошибки редки и нерегулярны, их трудно обнаруживать. Следовательно, *сокрытие имен желательно минимизировать*. Хотите проблем — используйте ***i*** или ***x*** в качестве глобальных имен, или в качестве локальных в функциях большого размера.

Доступ к скрытому глобальному имени осуществляется с помощью *операции* `::` (*операция разрешения области видимости — scope resolution operator*). Например:

```
int x;

void f2()
{
  int x=1;          // скрывает глобальную x
  ::x=2;            // присваивание глобальной x
  x=2;              // присваивание локальной x
  // ...
}
```

Не существует способа обращения к скрытому локальному имени.

Область видимости имени начинается в *точке объявления*, точнее, *сразу после декларатора*, но *перед инициализирующим выражением*. Из этого следует, что имя можно использовать в качестве инициализатора самого себя. Например:

```
int x;

void f3()
{
  int x = x;        // извращение: инициализируем x ее собственным
                    // (непроинициализированным) значением
}
```

Это не запрещено, просто глупо. Хороший компилятор предупредит о попытке чтения неинициализированной переменной (§5.9[9]).

В принципе, в блоке можно обращаться с помощью одного и того же имени к разным переменным, даже не используя операции `::`. Вот пример на эту тему:

```
int x=11;

void f4 ()
{
  int y = x;        // используется глобальная x: y=11
  int x = 22;
  y = x;            // используется локальная x: y=22
}
```

Считается, что *имена аргументов функции объявлены в самом внешнем блоке, относящемся к функции*, так что следующий пример

```
void f5 (int x)
{
  int x;            // error (ошибка)
}
```

компилируется с ошибкой, ибо имя x дважды определено в одной и той же области видимости. Отнесение такой ситуации к ошибочным позволяет избегать коварных ошибок.

4.9.5. Инициализация

Если инициализирующее выражение присутствует в объявлении, то оно задает начальное значение объекта. Если же оно отсутствует, то глобальные объекты (§4.9.4), объекты, объявленные в пространстве имен (§8.2) и статические локальные объекты (§7.1.2, §10.2.4) (совокупно называются *статическими объектами — static objects*) автоматически инициализируются нулевым значением соответствующего типа. Например:

```
int a;              // означает int a = 0;
double d;           // означает double d = 0.0;
```

Локальные переменные (иногда называемые также *автоматическими объектами — automatic objects*) и объекты, создаваемые в свободной памяти (*динамические объекты* или объекты, созданные на куче — *heap objects*), по умолчанию не инициализируются. Например:

```
void f()
{
  int x;            // x не имеет точно определенного значения
  // ...
}
```

Элементы массивов и члены структур инициализируются по умолчанию или не инициализируются в зависимости от того, являются ли соответствующие объекты статическими. Для пользовательских типов можно самостоятельно определить инициализацию по умолчанию (§10.4.2).

Сложные объекты требуют более одного инициализирующего значения. Для инициализации массивов (§5.2.1) и структур (§5.7) в стиле языка C используются *списки инициализирующих значений* (*initializer lists*), ограниченные фигурными скобками. Для инициализации пользовательских типов применяются конструкторы, которые используют обычный функциональный синтаксис, когда через запятую перечисляются их аргументы (§2.5.2, §10.2.3).

Обратите внимание на то, что в объявлениях пустая пара круглых скобок () всегда означает функцию (§7.1). Например:

```
int a[]={1, 2};   // инициализатор массива
Point z(1,2);     // инициализация в функциональном стиле (вызов конструктора)
int f();          // объявление функции
```

4.9.6. Объекты и леводопустимые выражения (lvalue)

Разрешено выделять память под неименованные объекты и использовать их, и можно осуществлять присваивания странно выглядящим выражениям (например, *p[a+10]=7). В результате возникает потребность в названии для «чего-то в памяти». Так мы приходим к фундаментальному определению объекта: *объект* — это непрерывный блок памяти. Соответственно, леводопустимое выражение (*lvalue*) — это выражение, ссылающееся на объект. Исходным смыслом lvalue (от английского left value) является «нечто, что может стоять в левой части операции присваивания». На самом деле, не каждое lvalue может стоять в левой части операции присваивания, ибо некоторые lvalue ссылаются на константы (§5.5). Те lvalue, которые были объявлены без модификаторов const, называются *модифицируемыми lvalue*. Приведенное простое и низкоуровневое понятие объекта не следует путать с высокоуровневыми понятиями классовых объектов и объектов полиморфных типов (§15.4.3).

Если программист не указал ничего иного (§7.1.2, §10.4.8), объявленный в функции объект создается в точке его определения и существует до момента выхода его имени из области видимости (§10.4.4). Такие объекты принято называть автоматическими объектами. Объекты, объявленные глобально или в пространствах имен, или с модификатором static в функциях или классах, создаются и инициализируются в работающей программе лишь однажды и живут до тех пор, пока программа не завершит работу (§10.4.9). Такие объекты принято называть статическими. Элементы массивов и нестатические члены классов и структур живут ровно столько, сколько живут содержащие их объекты.

Используя операции ***new*** и ***delete***, можно самостоятельно управлять временем жизни объектов (§6.2.6).

4.9.7. Ключевое слово typedef

Объявление, начинающееся с ключевого слова ***typedef***, *определяет новое имя для типа*, а не новую переменную какого-либо существующего типа. Например:

```
typedef char* Pchar;
Pchar p1, p2;        // p1 и p2 типа char*
char* p3=p1;
```

Вводимые таким образом имена часто являются короткими синонимами для типов с неудобоваримыми именами. Например, можно ввести для типа ***unsigned char*** (особенно при его частом использовании) *более короткий синоним*, ***uchar***:

```
typedef unsigned char uchar;
```

Часто ***typedef*** используется для того, чтобы ограничить прямое обращение к целевому типу единственным местом в программе. Например:

```
typedef int int32;
typedef short int16;
```

Если теперь использовать тип ***int32*** всюду, где нужны большие целые, то программу будет легко портировать на системную платформу, для которой ***sizeof(int)*** равен *2*. Действительно, нужно лишь дать новое определение для ***int32***:

```
typedef long int32;
```

Хорошо это или плохо, но с помощью ***typedef*** вводятся лишь новые синонимы существующих типов, а *не сами новые типы*. Следовательно, эти синонимы можно использовать параллельно именам существующих типов. Если же требуются новые типы, имеющие схожую семантику и/или представление, то следует присмотреться к перечислениям (§4.8) и классам (глава 10).

4.10. Советы

1. Сужайте область видимости имен; §4.9.4.
2. Не используйте одно и то же имя и в текущей области видимости, и в объемлющей области видимости; §4.9.4.
3. Ограничивайте объявление одним именем; §4.9.2.
4. Присваивайте короткие имена локальным и часто используемым переменным; глобальным и редко используемым — длинные; §4.9.3.
5. Избегайте похожих имен; §4.9.3.
6. Придерживайтесь единого стиля именования; §4.9.3.
7. Внимательно выбирайте имена таким образом, чтобы они отражали смысл, а не представление; §4.9.3.
8. Применяйте ***typedef*** для получения осмысленных синонимов встроенного типа в случаях, когда встроенный тип для представления переменных может измениться; §4.9.7.
9. Синонимы типов задавайте с помощью ***typedef***; новые типы вводите с помощью перечислений и классов; §4.9.7.
10. Помните, что любое объявление должно содержать тип явно (никаких «неявных ***int***»); §4.9.1.
11. Без необходимости не полагайтесь на конкретные числовые коды символов; §4.3.1, §C.6.2.1.
12. Не полагайтесь на конкретные размеры целых; §4.6.
13. Без необходимости не полагайтесь на диапазон значений чисел с плавающей запятой; §4.6.
14. Предпочитайте тип ***int*** типам ***short int*** и ***long int***; §4.6.
15. Предпочитайте тип ***double*** типам ***float*** и ***long double***; §4.5.
16. Предпочитайте тип ***char*** типам ***signed char*** и ***unsigned char***; §C.3.4.
17. Избегайте необязательных предположений о размерах объектов: §4.6.
18. Избегайте арифметики беззнаковых типов; §4.4.

19. Относитесь с подозрением к преобразованиям из ***signed*** в ***unsigned*** и обратным преобразованиям; §C.6.2.6.
20. Относитесь с подозрением к преобразованиям из типов с плавающей запятой в целые; §C.6.2.6.
21. Относитесь с подозрением к преобразованиям из более широких типов в более узкие (например, из ***int*** в ***char***); §C.6.2.6.

4.11. Упражнения

1. (*2) Запустите программу «Hello, world!» (§3.2). Если эта программа не компилируется, обратитесь к §B.3.1.
2. (*1) Для каждого объявления из §4.9 выполните следующее: если объявление не является определением, переделайте его в определение; если же объявление является определением, напишите для него соответствующее объявление, не являющееся одновременно и определением.
3. (*1.5) Напишите программу, которая выводит размеры фундаментальных типов, нескольких типов указателей и нескольких перечислений по вашему выбору. Используйте операцию ***sizeof***.
4. (*1.5) Напишите программу, которая выводит буквы '***a***' – '***z***' и цифры '***0***' – '***9***' и их десятичные коды. Повторите все для иных символов, имеющих зрительные образы. Выведите числовые коды в шестнадцатеричном виде.
5. (*2) Каковы на вашей машине минимальные и максимальные значения для следующих типов: ***char***, ***short***, ***int***, ***long***, ***float***, ***double***, ***long double*** и ***unsigned***?
6. (*1) Какова максимальная длина локальных имен на вашей машине? Какова максимальная длина для внешних имен на вашей машине? Есть ли ограничения на символы, которые можно использовать в именах?
7. (*2) Нарисуйте граф для фундаментальных типов, поддерживаемых любой стандартной реализацией (стрелка указывает направление от первого типа ко второму, если все значения первого типа представимы переменными второго типа). Нарисуйте тот же граф, но для вашей любимой реализации.

5

Указатели, массивы и структуры

Возвышенное и нелепое
часто столь тесно переплетены,
что их трудно рассматривать порознь.
— Том Пэйн

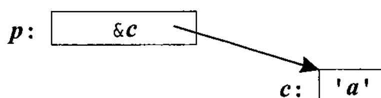
Указатели — нуль — массивы — строковые литералы — константы — указатели и константы — ссылки — ***void**** — структуры данных — советы — упражнения.

5.1. Указатели

Для заданного типа ***T***, тип ***T**** является «*указателем на **T***». Это означает, что переменные типа ***T**** *содержат адреса объектов типа **T***. Например:

```
char c = 'a';
char* p = &c;        // p содержит адрес c
```

или в графической форме:



К сожалению, *указатели на массивы* и *указатели на функции* требуют *более сложных конструкций объявления*:

```
int* pi;               // указатель на int
char** ppc;            // указатель на указатель на char
int* ap[15];           // массив из 15 указателей на int
int(*fp)(char*);       // указатель на функцию с аргументом char* и возвратом int
int* f(char*);         // функция с аргументом char* и возвратом "указатель на int"
```

Подробное объяснение этого синтаксиса дано в §4.9.1, а полное изложение грамматики — в Приложении А.

Основной операцией над указателями является *разыменование* (*dereferencing*), то есть обращение к объекту, на который показывает указатель. Эту операцию также называют *косвенным обращением* (*indirection*). Унарная операция разыменования обозначается знаком *, применяемым префиксным образом по отношению к операнду. Например:

```
char c = 'a';
char* p = &c;       // p содержит адрес переменной c
char c2 = *p;       // c2 == 'a'
```

Указатель ***p*** показывает на переменную ***c***, значением которой служит **'a'**. Таким образом, значение выражения ****p*** (присваиваемое ***c2***) есть **'a'**.

Допускается ряд арифметических операций над указателями на элементы массивов (§5.3). Указатели на функции могут быть чрезвычайно полезны; они обсуждаются в §7.7.

Указатели задуманы с целью непосредственного использования механизмов адресации памяти, реализуемых на машинном уровне. Большинство машин умеют адресовать отдельные байты. А те, что не могут, умеют извлекать байты из машинных слов. В то же время, редкие машины в состоянии адресовать отдельные биты. Следовательно, наименьший объект, который можно независимо разместить в памяти и которым можно манипулировать через встроенные указатели, есть объект типа ***char***. Стоит отметить, что тип ***bool*** требует по меньшей мере столько же памяти, что и тип ***char*** (§4.6). Для более компактного хранения малых объектов, можно использовать побитовые логические операции (§6.2.4) или битовые поля в структурах (§C.8.1).

5.1.1. Нуль

Нуль (***0***) имеет *тип* ***int***. Благодаря стандартным преобразованиям (§C.6.2.3), нуль можно использовать в качестве константы любого интегрального типа (§4.1.1), типа с плавающей запятой, указателя или указателя на член класса. Тип нуля определяется по контексту. Указатель со значением ***0*** называется *нулевым указателем* (*null pointer*) и обычно (но не всегда) представим на машинном уровне в виде последовательности нулевых битов соответствующей длины.

Не существует объектов в памяти с адресом нуль. В результате, *нуль служит литералом указательного типа*, означающим, что указатель с таким значением *не ссылается ни на какой объект*.

В языке С было принято использовать макроконстанту *NULL* для обозначения нулевых указателей. Из-за более плотного контроля типов в языке С++, использование простого ***0*** вместо так или иначе определенной макроконстанты *NULL* приводит к меньшим проблемам. Если вы чувствуете, что вам все же нужен *NULL*, воспользуйтесь

```
const int NULL = 0;
```

Модификатор ***const*** (§5.4) предотвращает случайное изменение *NULL* и гарантирует, что *NULL* можно использовать всюду, где требуется константа.

5.2. Массивы

Для заданного типа *T*, тип *T*[*size*] является «массивом из *size* элементов типа *T*». Индексация (нумерация) элементов требует индексов от *0* и до *size-1*. Например:

```
float v[3];          // массив из трех элементов типа float: v[0], v[1], v[2]
char* a[32];         // массив из 32 указателей на char: a[0] .. a[31]
```

Количество элементов массива (размер массива) должно задаваться константным выражением (§С.5). Если же нужен массив переменного размера, воспользуйтесь типом *vector* (§3.7.1, §16.3). Например:

```
void f(int i)
{
  int v1[i];             // ошибка: размер массива не константное выражение
  vector<int> v2(i);     // ok (все нормально)
}
```

Многомерные массивы определяются как массивы массивов. Вот соответствующий пример:

```
int d2[10][20];          // d2 есть массив из 10 массивов по 20 целых в каждом
```

Если здесь, как это принято во многих других языках программирования, отделять запятой различные размеры массива, то возникнет ошибка компиляции, ибо *запятая* в языке С++ обозначает *операцию следования* (*sequencing operator*, §6.2.2), и которая неприменима в константных выражениях (§С.5). Можете проверить это на следующем примере:

```
int bad[5,2];            // error: запятая недопустима в константных выражениях
```

Многомерные массивы рассматриваются в §С.7. В коде высокого уровня их лучше вообще избегать[1].

5.2.1. Инициализация массивов

Массив можно проинициализировать списком значений. Например:

```
int  v1[] = {1,2,3,4};
char v2[] = {'a','b','c',0};
```

Когда массив объявляется без явного указания размера, но со списком инициализирующих значений, *размер вычисляется по количеству этих значений*. Следовательно, *v1* и *v2* имеют типы *int*[*4*] и *char*[*4*], соответственно. Когда размер массива указан явно, а в инициализирующем списке присутствует большее количество значений, то возникает ошибка компиляции. Например:

```
char v3[2] = { 'a', 'b', 0 };  // error: слишком много инициализаторов
char v4[3] = { 'a', 'b', 0 };  // ok
```

Если же в инициализирующем списке значений не хватает, то «оставшимся не у дел» элементам массива присваивается *0*. Например:

---

[1] Трудно согласиться с данным тезисом автора — разве матричная математика не относится к высокоуровневому коду? — *Прим. ред.*

```
int v5[8] = {1,2,3,4};
```

Эквивалентно

```
int v5[] = {1,2,3,4,0,0,0,0};
```

Заметим, что *присваивания списком значений для массивов не существует*:

```
void f()
{
  v4 = {'c','d',0}; // error: такое присваивание для массивов недопустимо
}
```

Если нужно подобное присваивание, воспользуйтесь стандартными типами **vector** (§16.3) или **valarray** (§22.4).

Массив символов удобно инициализировать строковыми литералами (§5.2.2).

5.2.2. Строковые литералы

Строковым литералом (*string literal*) называется последовательность символов, заключенная в *двойные кавычки*.

```
"this is a string"
```

Строковые литералы содержат *на один символ больше*, чем это кажется на первый взгляд: они оканчиваются нулевым символом **'0'**, числовое значение которого равно нулю. Например:

```
sizeof("Bohr")==5
```

Тип строкового литерала есть «массив соответствующего количества константных символов», так что литерал **"Bohr"** имеет тип **const char[5]**.

Строковый литерал допускается присваивать указателю типа ***char****, так как в более ранних версиях языков C и C++ типом строковых литералов был ***char****. Благодаря этому допущению миллионы строк ранее написанного на C и C++ кода остаются синтаксически корректными. Тем не менее, попытка модификации строкового литерала через такой указатель ошибочна:

```
void f()
{
  char* p = "Plato";
  p[4] = 'e';          // error: присваивание константе; результат не определен
}
```

Такого рода ошибки обычно не выявляются до стадии выполнения программы, и к тому же, разные реализации по-разному реагируют на нарушение этого правила (см. также §B.2.3). Очевидность константного характера строковых литералов позволяет компиляторам оптимизировать как методы хранения этих литералов, так и методы доступа к ним.

Если нам нужна строка с гарантированной возможностью модификации, то нужно скопировать символы в массив:

```
void f()
{
  char p[] = "Zeno"; // p есть массив из 5 char
```

```
  p[0] = 'R';          // ok
}
```

Память под строковые литералы выделяется *статически*, поэтому их возврат из функций *безопасен*. Например:

```
const char* error_message(int i)
{
  //...
  return "range error";
}
```

Память, содержащая строку **"range error"**, никуда не денется после выходя из функции **error_message()**.

От конкретной реализации компилятора зависит, будут ли два одинаковых строковых литерала сосредоточены в одном и том же месте памяти (§C.1). Например:

```
const char* p = "Heraclitus";
const char* q = "Heraclitus";

void g()
{
  if(p==q) cout<<"one!\n";   // результат зависит от конкретной реализации
}
```

Обратите внимание на то, что для указателей операция == сравнивает адреса (значения указателей), а не адресуемые ими величины.

Пустая строка (*empty string*) записывается в виде *смежных двойных кавычек* ("") и имеет тип **const char**[*1*].

Синтаксис, использующий обратную косую черту для обозначения специальных символов (§C.3.2), применим и в строковых литералах. Это позволяет вносить в содержимое строковых литералов и двойные кавычки ("), и символ обратной косой черты (\). Чаще всего, конечно, используется символ **'\n'** для принудительного перевода строки вывода. Например:

```
cout<<"beep at the end of message\a\n";
```

Символ **'a'** из ASCII-таблицы называется ***BEL*** (звонок); он приводит к подаче звукового сигнала.

Переходить же на новую строку исходного текста программы при записи содержимого строкового литерала *нельзя*:

```
"this is not a string
but a syntax error"
```

С целью улучшения читаемости текста программы длинные строковые литералы можно разбить на фрагменты, отделяемые друг от друга лишь пробельными символами[1]. Например:

```
char alpha[] = "abcdefghijklmnopqrstuvwxyz"
               "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
```

[1] В том числе, и переходом на новую строку текста программы. — *Прим. ред.*

Компилятор объединяет смежные строковые литералы в один литерал, так что переменную alpha можно было бы с тем же успехом инициализировать единственной строкой:

```
"abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ";
```

В состав строк можно включать нулевые символы, но большинство функций и не догадаются о том, что после нулей в них есть что-то еще. Например, строка **"Jens\000Munk"** будет трактоваться как **"Jens"** такими библиотечными функциями, как **strcpy**() или **strlen**() (см. §20.4.1).

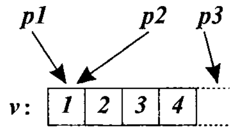
Строки с префиксом **L**, например **L"angst"**, состоят из символов типа **wchar_t** (§4.3, §C.3.3). Тип этих строк есть **const wchar_t**[].

5.3. Указатели на массивы

В языке C++ *указатели и массивы тесно связаны. Имя массива* может быть использовано в качестве *указателя на его первый элемент*. Например:

```
int v[]={1,2,3,4};
int* p1=v;          // указатель на начальный элемент (неявное преобразование)
int* p2=&v[0];      // указатель на начальный элемент
int* p3=&v[4];      // указатель на элемент, следующий за последним
```

или в графической форме:



Гарантируется возможность настройки указателя на «*элемент, расположенный за последним элементом массива*». Это важный момент для многих алгоритмов (§2.7.2, §18.3). Однако поскольку такой указатель на самом деле не указывает ни на какой элемент массива, его нельзя (не стоит) использовать ни для записи, ни для чтения. Возможность получения *осмысленного адреса «элемента, расположенного перед первым элементом массива», не гарантируется* и таких действий следует избегать. В рамках некоторых машинных архитектур память под массивы может выделяться в самом начале адресного пространства, так что адреса «элемента, расположенного перед первым элементом массива» может просто не существовать.

Неявные преобразования имени массива в указатель на его первый элемент широко используются *при вызовах функций в C-стиле*. Например:

```
extern "C" int strlen(const char*);  // из <string.h>

void f()
{
  char v[] = "Annemarie";
  char* p = v;                  // неявное преобразование от char[] к char*
  strlen(p);
  strlen(v);                    // неявное преобразование от char[] к char*
  v = p;                        // error: неверное присваивание массиву
}
```

Здесь в обоих вызовах стандартной библиотечной функции ***strlen*** () передается одно и то же адресное значение. Заковыка в том, что избежать такого неявного преобразования нельзя, так как невозможно объявить функцию, при вызове которой копировался бы сам массив ***v***. К счастью, не существует обратного (явного и неявного) преобразования от указателя к массиву.

Неявное преобразование от массива к указателю, имеющее место при вызове функции, означает, что при этом *размер массива теряется*. Ясно, что, тем не менее, функция должна каким-то образом узнать размер массива, иначе невозможно будет выполнить осмысленные действия над ним. Как и другие функции из стандартной библиотеки языка C, получающие указатель на символьные строки, функция ***strlen*** () полагается на *терминальный нуль* в качестве *индикатора конца строки*; ***strlen*** (***p***) возвращает количество символов в строке вплоть до, но не считая терминального ***нуля***. Все это достаточно низкоуровневые детали. Типы ***vector*** (§16.3) и ***string*** (глава 20) из стандартной библиотеки не страдают этими проблемами.

5.3.1. Доступ к элементам массивов

Эффективный и элегантный доступ к элементам массивов (и другим структурам данных) является ключевым фактором для многих алгоритмов (§3.8, глава 18). Такой доступ можно осуществлять либо через указатель на начало массива и индекс элемента, либо непосредственно по указателю на элемент. Например, следующий код, применяющий индексы для прохода по символам строки

```
void fi (char v[ ] )
{
  for (int i = 0; v[i] !=0; i++)
    // используется v[i]
}
```

эквивалентен коду, применяющему с той же целью указатель:

```
void fp (char v[ ] )
{
  for (char* p = v; *p!=0; p++)
    // используется *p;
}
```

Унарная операция разыменования (операция *) действует так, что ****p*** есть символ, адресуемый указателем ***p***, а операция ++ *инкрементирует значение указателя* с тем, чтобы он показывал на *следующий элемент массива*.

Не существует фундаментальных причин, по которым одна из этих версий кода была бы эффективнее другой. Современные компиляторы должны превращать оба варианта в одинаковые фрагменты машинного кода (§5.9[8]). Программисты вольны осуществлять выбор между ними, исходя из логических или эстетических соображений.

Результат воздействия на указатели арифметических операций +, -, ++ и — зависит от типа объектов, на которые указатель ссылается. Когда арифметическая операция применяется к указателю p типа ***T****, предполагается, что p указывает на элемент массива объектов типа ***T***; ***p+1*** указывает на следующий элемент этого массива, а ***p-1*** — на предыдущий элемент. Это подразумевает, что адресное (целочисленное)

значение выражения *p+1* на *sizeof(T)* больше, чем величина указателя *p*. Например, результатом выполнения кода

```
#include <iostream>

int main ()
{
  int vi[10];
  short vs[10];

  std::cout<< &vi[0] <<' '<<&vi[1]<< '\n';
  std::cout<< &vs[0] <<' '<<&vs[1]<<'\n';
}
```

будет

```
0x7fffaef0   0x7fffaef4
0x7fffaedc   0x7fffaede
```

при использовании принятой по умолчанию шестнадцатеричной формы записи значений указателей. Из данного примера видно, что в моей реализации *sizeof(short)* есть *2*, а *sizeof(int)* есть *4*.

Вычитание указателей друг из друга определено для случая, когда *оба указателя показывают на элементы одного и того же массива* (язык, правда, не предоставляет быстрых способов проверки этого факта). Результатом вычитания указателей будет *количество элементов массива* (целое число), расположенных между указуемыми элементами. К указателю можно прибавить или вычесть целое; в обоих случаях результатом будет указатель. При этом, если полученный таким образом указатель не указывает на один из элементов того же массива (или на элемент, расположенный за последним элементом массива), то результат его применения не определен. Например:

```
void f()
{
  int v1[10];
  int v2[10];

  int i1 = &v1[5]-&v1[3];   // i1 = 2
  int i2 = &v1[5]-&v2[3];   // результат не определен

  int* p1 = v2+2;           // p1 = &v2[2];
  int* p2 = v2-2;           // *p2 не определен
}
```

Обычно, в использовании более-менее сложной арифметики указателей необходимости нет, и ее лучше избегать. *Сложение указателей не имеет смысла вообще*; в итоге оно *запрещено.*

Массивы не являются самоописываемыми сущностями, так как их размеры не хранятся вместе с ними. Это предполагает, что для успешного перемещения по элементам массива нужно так или иначе заполучить его размер. Вот пример на эту тему:

```
void fp (char v [ ] , unsigned int size)
{
 for (int i=0; i<size; i++)
   // используется v[i]

 const int N = 7;
 char v2 [N] ;
 for (int i=0; i<N; i++)
   // используется v2[i]
}
```

Заметим, что большинство реализующих язык C++ программных средств не предоставляет никаких способов контроля границ массива. Такая концепция массивов весьма низкоуровневая. Более интеллектуальные концепции массивов можно реализовать с помощью классов (§3.7.1).

5.4. Константы

В языке C++ реализована концепция определяемых пользователем констант, явно отражающая тот факт, что значение нельзя изменять непосредственно. Это полезно во многих отношениях. Например, существуют объекты, которые не требуют изменений после их инициализации; использование символических констант обеспечивает более удобное сопровождение кода, чем в случае непосредственного применения литералов в коде программы; указатели часто используются только для чтения, а не для записи; большинство параметров функций предназначены лишь для чтения, а не для перезаписи.

Ключевое слово ***const*** используется для объявления константного объекта. Так как константным объектам присваивать значения нельзя, то они в *обязательном порядке* должны инициализироваться. Например:

```
const int model = 90;           // model является константой
const int v [ ] = {1,2,3,4} ;   // v[i] являются константами

const int x;                    // error: нет инициализатора
```

Объявление объекта константным гарантирует, что значение объекта не изменится в его области видимости:

```
void f ( )
{
 model = 200;                   // error
 v [2] ++;                      // error
}
```

Заметим, что ключевое слово ***const*** модифицирует именно *тип*, то есть оно ограничивает способы использования объекта, а *не его размещение в памяти*. Например:

```
void g (const X* p)
{
 // здесь нельзя модифицировать *p
}
```

```
void h ()
{
  X val;                    // val можно изменять
  g (&val) ;
  // ...
}
```

В зависимости от своего «интеллекта», конкретные компиляторы могут по-разному использовать факт константности объекта. Например, инициализатор константного объекта часто (но не всегда) является константным выражением (§С.5); если это так, то его можно вычислить во время компиляции. Далее, если компилятор может выявить все случаи использования константы, то он может не выделять под нее память. Например:

```
const int c1 = 1;
const int c2 = 2;
const int c3 = my_f(3) ;  // значение c3 не известно в момент компиляции
extern const int c4;      // значение c4 не известно в момент компиляции
const int* p = &c2;       // под c2 нужно выделить память
```

Так как компилятор знает значения переменных ***c1*** и ***c2***, он может использовать их в константных выражениях. Поскольку значения для ***c3*** и ***c4*** во время компиляции неизвестны (исходя лишь из информации в данной единице компиляции; см. §9.1), для них требуется выделить память. Адрес переменной ***c2*** вычисляется (и где-то, наверное, используется), и для нее память выделять тоже нужно. Самым простым и часто встречающимся случаем является ситуация, когда *значение константы во время компиляции известно и память под нее не выделяется*; ***c1*** служит примером такого случая. Ключевое слово ***extern*** указывает, что переменная ***c4*** определена *где-то в другом месте программы* (§9.2).

Как правило, для массива констант память всегда выделяется, так как компилятор не в состоянии выявить, какие именно элементы массива используются в выражениях. Впрочем, на многих машинах и в этом случае можно достичь увеличения эффективности за счет помещения таких массивов в области памяти с атрибутом «только для чтения».

Константы часто используются в качестве размера массивов и в качестве меток case-ветвей оператора ***switch***. Например:

```
const int a = 42;
const int b = 99;
const int max = 128;

int v[max] ;

void f(int i)
{
  switch (i)
  {
    case a:
      // ...
    case b:
      // ...
  }
}
```

В подобного рода случаях часто альтернативой константам служат перечисления (§4.8).

Как ключевое слово const применяется к функциям-членам классов, рассказывается в §10.2.6 и §10.2.7.

Символические константы следует использовать систематически, дабы избегать появления «магических чисел» в программном коде. Если такое число обозначает, например, размер массива, то будучи многократно продублированным непосредственно в коде, оно вызовет серьезные затруднения при модификации программы, так как потребуется выявить и правильно изменить каждое его вхождение. Использование символических констант локализует информацию. Часто числовые константы означают некоторые предположения о работе программы. Например, число *4* может означать количестве байт в целом типе, *128* — емкость буфера ввода, а *6.24* — обменный курс датской кроны по отношению к доллару США. Тому, кто впоследствии сопровождает программу, понять смысл числовых литералов бывает довольно трудно. Часто на такие числа не обращают должного внимания при переносе программ в иные среды, или при иных модификациях программ, нарушающих первоначальные предположения, что в итоге приводит к весьма неприятным ошибкам. Хорошо прокомментированные символические константы сводят указанные проблемы к минимуму.

5.4.1. Указатели и константы

В операциях с указателями участвуют сразу два объекта: указатель и объект, на который он ссылается. *Помещение ключевого слова const в начало объявления делает константным объект*, а *не указатель*. Для объявления указателя константой нужно применить декларатор ***const** вместо просто звездочки. Например:

```
void f1 (char* p)
{
  char s[] = "Gorm";

  const char* pc = s;          // указатель на константу
  pc[3] = 'g';                 // error: pc указывает на константу
  pc = p;                      // ok

  char* const cp = s;          // константный указатель
  cp[3] = 'a';                 // ok
  cp = p;                      // error: cp есть константа

  const char* const cpc = s;   // константный указатель на константу
  cpc[3] = 'a';                // error: cpc указывает на константу
  cpc = p;                     // error: cpc есть константа
}
```

Подчеркнем, что константой указатель делает именно декларатор ***const**. Никакого декларатора **const*** не существует, так что ключевое слово **const**, расположенное перед звездочкой, относится к базовому типу. Например:

```
char* const cp;               // константный указатель на char
char const* pc;               // указатель на константу типа char
const char* pc2;              // указатель на константу типа char
```

Некоторые люди находят *удобным читать такие объявления справа налево.* Например, «***cp*** это константный указатель на ***char***», или «***pc2*** есть указатель на ***char*** *константный*».

Объект может быть константным при обращении к нему через некоторый указатель, и обычной переменной при иных способах доступа. Это широко используется в контексте параметров функций. Если у функции параметр-указатель объявлен *константным*, то функция лишается возможности изменять адресуемое им значение. Например:

```
char* strcpy(char* p, const char* q); // не может модифицировать *q
```

Вы можете присвоить адрес переменной указателю на константу, поскольку от такого присваивания не будет никакого вреда. Противоположное неверно: нельзя присвоить адрес константы обычному (неконстантному) указателю, ибо в противном случае с его помощью можно было бы изменить значение константы. Например:

```
void f4()
{
  int a = 1;
  const int c = 2;
  const int* p1 = &c;        // ok
  const int* p2 = &a;        // ok

  int* p3 = &c;              // error: инициализация переменной типа int*
                             // значением типа const int*
  *p3 = 7;                   // попытка изменить значение константы c
}
```

Снять ограничения на применение указателей на константы можно с помощью явного приведения типа (§10.2.7.1 и §15.4.2.1).

5.5. Ссылки

Ссылка (*reference*) является альтернативным именем объекта. Ссылки чаще всего используются для объявления параметров и возвращаемых значений у функций вообще, и у перегруженных операций (глава 11) в частности. Обозначение ***X&*** означает *ссылку на* ***X*** (*reference to X*). Например:

```
void f()
{
  int i = 1;
  int& r = i;                // r и i теперь ссылаются на одно и то же целое
  int x = r;                 // x = 1
  r = 2;                     // i = 2
}
```

Чтобы гарантировать привязанность ссылки к некоторому объекту, мы обязаны ее инициализировать. Например:

```
int i = 1;
int& r1 = i;                 // ok: r1 инициализировано
```

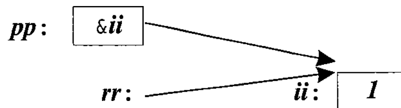
```
int& r2;                    // error: отсутствует инициализатор
extern int& rr3;            // ok: r3 инициализируется в другом месте
```

Инициализация ссылок это не то же самое, что присваивание. Несмотря на обманчивую видимость, *ни одна операция не выполняется над самой ссылкой.* Например:

```
void g ()
{
  int ii = 0;
  int& rr = ii;
  rr++;                     // ii увеличивается на 1
  int* pp = &rr;            // pp указывает на ii
}
```

Здесь все выражения допустимы, но в результате *r*++ увеличивается не ссылка rr, а адресуемая этой ссылкой переменная ***ii*** целого типа. Следовательно, *значение ссылки никогда не меняется после инициализации*; она всегда ссылается на объект, с которым была проинициализирована. Чтобы получить указатель на объект, с которым связана ссылка ***rr***, можно использовать выражение &***rr***.

Очевидной гипотетической реализацией ссылки может служить (константный) указатель, который автоматически (неявно) разыменуется при использовании. Большого вреда в такой интерпретации нет, особенно если всегда помнить, что над ссылками нельзя выполнять операции так же, как над указателями:



В ряде случаев компилятор может так оптимизировать код, что в момент исполнения программы ей не будет соответствовать никакого объекта в памяти.

Инициализация ссылок тривиальна, когда инициализатор представляет из себя ***lvalue*** (объект, адрес которого можно получить; §4.9.6). Инициализатором для типа ***T***& должен быть ***lvalue*** типа ***T***.

Инициализатор для типа ***const T***& не обязан следовать этому правилу и даже может вообще не иметь тип ***T***. В таких случаях:

1. Если необходимо, выполняется неявное преобразование к типу ***T*** (§C.6).
2. Результирующее значение помещается во временную переменную типа ***T***.
3. Временная переменная используется в качестве инициализатора.

Рассмотрим пример:

```
double& d = 1;              // error: требуется lvalue
const double& cdr = 1;      // ok
```

Можно интерпретировать последнюю инициализацию следующим образом:

```
double temp = double (1) ;  //создаем временную переменную с нужным значением
const double& cdr = temp;   //затем используем ее для инициализации cdr
```

Временная переменная, созданная для хранения инициализатора ссылки, существует до тех пор, пока ссылка не выйдет из своей области видимости.

Видно, что ссылки на константы и ссылки на переменные серьезно различаются, так как введение временных переменных в последнем случае приводило бы к неприятным ошибкам: изменение исходной переменной влечет за собой изменение быстро исчезающей временной переменной. В случае ссылок на константы таких проблем нет. Ссылки на константы находят широкое применение в качестве аргументов функций (§11.6).

Ссылки можно использовать для объявления аргументов функций с тем, чтобы функции могли изменять значения передаваемых им объектов. Например:

```
void increment (int& aa) { aa++; }

void f()
{
  int x = 1;
  incremen (x);      // x = 2
}
```

Семантика передачи аргументов аналогична инициализации, так что при вызове функции ***increment***() аргумент ***aa*** *становится другим именем* для ***x***. Для улучшения читаемости программ лучше избегать функций, изменяющих свои аргументы. Предпочтительнее использовать явным образом возврат функции или объявить аргумент указателем:

```
int next (int p) {return p+1; }
void incr (int* p) { (*p) ++; }

void g ()
{
  int x = 1;
  incremen (x);      // x = 2
  x = next (x);      // x = 3
  incr (&x);         // x = 4
}
```

Внешний вид вызова ***increment***(***x***) не дает читателю и намека на то, что значение ***x*** будет изменено, в отличие от ***x*** = ***next***(***x***) и ***incr***(&***x***). Так что ссылочные аргументы уместно использовать лишь в функциях, чьи имена прозрачно намекают на возможность модификации аргументов.

Ссылки также применяются для определения функций, которые можно использовать и в правой, и в левой частях операции присваивания. Как всегда, наиболее интересные примеры, демонстрирующие столь хитрое поведение функций, сосредоточены в рамках нетривиальных пользовательских типов. Давайте для примера определим сейчас простой ассоциативный массив. Сначала задаем структуру ***Pair*** следующим образом:

```
struct Pair
{
  string name;
  double val;
};
```

Общая идея заключается в том, что строка имеет ассоциированное с ней числовое значение с плавающей запятой. Несложно определить функцию ***value***(), кото-

рая поддерживает структуру данных, содержащую ровно один элемент типа ***Pair*** для каждой уникальной строки, переданной этой функции. Чтобы не затягивать рассмотрение вопроса, приведем очень простую (и неэффективную) реализацию:

```
vector<Pair> pairs;

double& value(const string& s)
/*
  поддерживает набор пар Pair:
  ищет строку s; если найдена, возвращает соответствующее значение;
  в противном случае создает новый Pair и возвращает 0. См. также §11.8.
*/
{
  for(int i = 0; i <pairs.size(); i++)
    if(s == pairs[i].name) return pairs[i].val;

  Pair p = {s,0};
  pairs.push_back(p);      // добавить Pair в конец (§3.7.3)
  return pairs[pairs.size()-1].val;
}
```

Здесь речь идет о построении *массива чисел, индексируемых строками*. Для заданного строкового аргумента функция ***value***() находит соответствующий числовой объект (именно объект, а не его значение) и возвращает ссылку на него. Вот пример использования функции ***value***():

```
int main()          // посчитать кол-во вхождений каждого слова во входном потоке
{
  string buf;

  while(cin>>buf) value(buff)++;
  for(vector<Pair>::const_iterator p = pairs.begin(); p!=pairs.end(); ++p)
       cout<< p->name << ": " << p->val << '\n';
}
```

Здесь в каждой итерации цикла ***while*** из стандартного потока ввода ***cin*** читается одно слово и записывается в строковый буфер ***buf*** (), после чего обновляется значение связанного с этим словом счетчика. В конце печатается результирующая таблица введенных слов вместе с числами их повторов. Например, если на входе были слова

```
aa  bb  bb  aa  aa  bb  aa  aa
```

то программа выдаст следующую статистическую информацию:

```
aa: 5
bb: 3
```

Несложно переделать представленное решение в настоящий ассоциативный массив на базе классового шаблона с перегруженной операцией [] (§11.8). Еще проще воспользоваться типом ***map*** из стандартной библиотеки (§17.4.1).

5.6. Тип void*

Переменной типа ***void**** *можно присвоить значение указателя любого типа*; одной переменной типа ***void**** можно присваивать значение другой переменной этого типа, а также сравнивать их между собой на равенство (или неравенство); тип ***void**** можно явным образом преобразовывать в указатели иных типов. Другие операции с типом ***void**** опасны, так как компилятор не знает типа адресуемых объектов, и поэтому такие операции вызывают ошибку компиляции. Чтобы воспользоваться указателем типа ***void****, его нужно явно преобразовать к иным типам указателей. Например:

```
void f(int* pi)
{
  void* pv = pi;          // ok: неявное преобразование из int* в void*
  *pv;                    // error: нельзя разыменовать void*
  pv++;                   // error: нельзя инкрементировать void*
                          // (неизвестен размер указуемого объекта)

  int* pi2 = static_cast<int*> (pv) ;          // явное преобразование в int*

  double* pd1 = pv;                            // error
  double* pd2 = pi;                            // error
  double* pd3 = static_cast<double*> (pv) ;    // небезопасно
}
```

В общем случае небезопасно использовать указатель, приведенный к типу, отличному от типа адресуемого объекта. К примеру, под тип double на многих машинах отводится 8 байт. Тогда работа с указателем ***pd3*** будет непредсказуемой, так как ***pi*** указывает на блок памяти, выделенный под ***int*** (обычно 4 байта). *Операция* ***static_cast*** для явного преобразования типов указателей *внутренне опасна* и *внешне выглядит «безобразно»*, но так и было задумано, чтобы своим внешним видом она напоминала о реальной опасности преобразования.

В основном, тип ***void**** применяется в параметрах функций, которым не позволено делать предположения о типе адресуемых объектов, и для возврата из функций «бестиповых» объектов. Чтобы воспользоваться таким объектом, нужно явно преобразовать тип указателя.

Функции, использующие ***void****, *типичны для самых нижних слоев системного обеспечения*, где ведется работа с аппаратными ресурсами компьютера. Например:

```
void* my_alloc (size_t n) ;  // выделить n байт из моей специальной кучи
```

К наличию ***void**** на более высоких уровнях программного обеспечения следует относиться с подозрением — скорее всего это следствие ошибок проектирования. Если ***void**** используется для оптимизации, то его лучше скрыть за фасадом безопасного интерфейса (§13.5, §24.4.2).

Указатели на функции (§7.7) и *указатели на члены классов* (§15.5) *не могут присваиваться переменным типа* ***void****.

5.7. Структуры

Массив — это агрегат (набор) элементов одинакового типа. А *структура* — это *агрегат элементов* (почти) *произвольных типов*. Например:

```
struct address
{
  char* name;           // "Jim Dandy"
  long int number;      // 61
  char* street;         // "South St"
  char* town;           // "New Providence"
  char state [2] ;      // 'N' 'J'
  long zip;             // 7974
};
```

Здесь определяется *новый тип с именем* ***address***, состоящий из полей, которые необходимо заполнить для отправки письма. Обратите внимание на *точку с запятой в конце определения*. Для языка C++ это довольно редкий случай, когда после закрывающей фигурной скобки нужно ставить точку с запятой, так что люди часто забывают о ней, а это вызывает *ошибку компиляции*.

Переменные типа ***address*** можно объявлять точно так же, как и другие переменные, а индивидуальные поля (*члены — members*) достижимы с помощью операции . (*операция точка — dot operator*). Например:

```
void f()
{
  address jd;
  jd.name = "Jim Dandy";
  jd.number = 61;
}
```

Конструкция, использованная нами ранее для инициализации массивов (§5.2.1) применима и для инициализации переменных структурных типов. Например:

```
address jd = {"Jim Dandy", 61, "South St",
              "New Providence", {'N','J'}, 7974};
```

Впрочем, конструкторы обычно подходят лучше (§10.2.3). Заметьте, что ***jd.state*** нельзя инициализировать строкой "***NJ***". Строки оканчиваются терминальным символом '***0***'. Следовательно, "***NJ***" содержит три символа — на один больше, чем можно разместить в ***jd.state***.

К объектам структурных типов часто обращаются с помощью указателей, используя операцию -> (*операция разыменования указателей на структуры — structure pointer dereference operator*). Например:

```
void print_addr(address* p)
{
  cout<< p->name << '\n'
      << p->number << ' ' << p->street << '\n'
      << p->town << '\n'
      << p->state[0] << p->state[1] << ' ' << p->zip<< '\n';
};
```

Для указателя p выражение *p->m эквивалентно выражению (*p).m.*

Объекты структурных типов можно присваивать, передавать в функции и *возвращать из функций.* Например:

```
address current;

address set_current (address next)
{
  address prev = current;
  current = next;
  return prev;
}
```

Другие операции, например операции сравнения (== и !=), *не определены.* Однако пользователь (программист) *может определить их сам* (глава 11).

Размер объекта структурного типа не обязательно равен сумме размеров полей структуры, так как для разных машинных архитектур нужно в обязательном порядке или целесообразно (для эффективности) располагать объекты определенных типов, выровненными по заданным границам. Например, целые часто располагаются на границах машинных слов. При этом говорят, что объекты надлежащим образом *выровнены* (*aligned*) в памяти компьютера. Это приводит к «дырам» в структурах. Например, для многих машин окажется, что ***sizeof*(*address*)** равен ***24***, а не ***22***, как можно было бы ожидать. Можно попытаться минимизировать неиспользуемое пространство, отсортировав поля по убыванию их размера. Однако все же лучше упорядочивать поля структур из соображений наглядности, а сортировку по размеру производить лишь в случае жестких требований по оптимизации.

Имя нового типа можно использовать сразу же после его появления, а вовсе не после его полного определения. Например:

```
struct Link
{
  Link* previous;
  Link* successor;
};
```

До окончания полного определения типа нельзя объявлять объекты или поля этого типа. Например:

```
struct No_good
{
  No_good member;  // error: рекурсивное определение
};
```

Тут возникает ошибка компиляции, ибо *компилятор не знает, сколько памяти нужно отвести под* ***No_good***. Чтобы позволить двум (и более) структурным типам ссылаться друг на друга, достаточно объявить, что некоторое имя является именем структурного типа. Например:

```
struct List;          // определение будет дано ниже

struct Link
{
  Link* pre;
```

```
  Link* suc;
  List* member_of;
};

struct List
{
  Link* head;
};
```

*Без предварительного объявления имени **List** использование этого имени* в объявлении структуры ***Link** привело бы к ошибке компиляции.*

Именем структурного типа можно пользоваться до его полного определения в случаях, когда нет обращений к полям структуры или нет необходимости знать размер типа. Например:

```
class S;            // S - это имя некоторого типа

extern S a;
S f();
void g(S);
S* h(S*);
```

Тем не менее, многим объявлениям требуется полное определение типа:

```
void k(S* p)
{
  S a;              // error: S не определен и его размер неизвестен
  f();              // error: S не определен, а его размер нужен
                    // для возвращаемого значения

  g(a);             // error: S не определен, а его размер нужен для передачи аргумента
  p->m = 7;         // error: S не определен и его члены неизвестны
  S* q = h(p);      // ok: с указателями проблем нет
  q->m = 7;         // error: S не определен и его члены неизвестны
}
```

Структуры являются упрощенными классами (глава 10).

По причинам, уходящим корнями глубоко в предысторию языка C, разрешается совмещать имена структурного типа и иных программных конструкций в рамках одной и той же области видимости. Например:

```
struct stat{ /* ... */ };
int stat(char* name, struct stat* buf);
```

В этом случае, просто имя ***stat*** относится к неструктурным конструкциям, а имя структуры должно предваряться ключевым словом ***struct***. Аналогично, требуется использовать ключевые слова ***class***, ***union*** (§C.8.2) и ***enum*** (§4.8) с целью преодоления возникающих неоднозначностей. Лучше, однако, избегать такого конфликта имен.

5.7.1. Эквивалентность типов

Две структуры являются разными типами, даже если у них поля одинаковые. Например, структуры

```
struct S1 { int a; };
struct S2 { int a; };
```

являются разными типами, так что следующий код ошибочен:

```
S1 x;
S2 y = x;        // error: несоответствие типов
```

Структуры разнятся также и от фундаментальных типов:

```
S1 x;
int i = x;        // error: несоответствие типов
```

Каждая *структура* должна иметь единственное определение в программе (§9.2.3).

5.8. Советы

1. Избегайте нетривиальной арифметики указателей; §5.3.
2. Внимательно отслеживайте потенциальную возможность выхода за границы массива; §5.3.1.
3. Используйте ***0*** вместо ***NULL***; §5.1.1.
4. Используйте типы ***vector*** и ***valarray*** вместо встроенных (в С-стиле) массивов; §5.3.1.
5. Используйте ***string***, а не массивы ***char*** с терминальным нулем; §5.3.
6. Минимизируйте применение простых ссылок для аргументов функций; §5.5.
7. Применяйте ***void**** лишь в низкоуровневом коде; §5.6.
8. Избегайте нетривиальных литералов («магических чисел») в коде — используйте символические макроконстанты; §4.8, §5.4.

5.9. Упражнения

1. (*1) Напишите следующие объявления: указатель на символ, массив из 10 целых, ссылка на массив из 10 целых, указатель на массив строк, указатель на указатель на символ, целая константа, указатель на целую константу, константный указатель на целое. Проинициализируйте все объекты.
2. (*1.5) Каковы на вашей системе ограничения на значения указателей типа ***char****, ***int**** и ***void****? Например, может ли ***int**** иметь нечетное значение (подсказка: подумайте о выравнивании)?
3. (*1) Используйте ***typedef*** для определения типов: ***unsigned char***, ***const unsigned char***, указатель на целое, указатель на указатель на ***char***, указатель на массив ***char***, массив из 7 указателей на ***int***, указатель на массив из 7 указателей на ***int*** и массив из 8 массивов по 7 указателей на целые.

4. (*1) Напишите функцию, которая обменивает значения у двух целых чисел. Используйте аргументы типа ***int****. Напишите другой вариант с аргументами типа ***int***&.

5. (*1.5) Каков размер массива ***str*** в следующем примере:

   char str [ ] = "***a short string***" ;

   Какова длина строки "***a short string***"?

6. (*1) Определите функции ***f***(***char***), ***g***(***char***&) и ***h***(***const char***&). Вызовите их с аргументами '***a***', ***49***, ***3300***, ***c***, ***uc*** и ***sc***, где ***c*** — это ***char***, ***uc*** — ***unsigned char***, ***sc*** — ***signed char***. Какие из этих вызовов допустимы? В каких вызовах компилятор создаст временные переменные?

7. (*1.5) Определите таблицу имен месяцев и количества дней в них. Выведите эту таблицу. Проделайте это дважды: первый раз воспользуйтесь массивом элементов типа ***char*** для названия месяца и массивом типа ***int*** для количества дней; второй раз примените массив структур, которые содержат и названия месяцев, и число дней.

8. (*2) Выполните тестовые прогоны, чтобы на практике убедиться в эквивалентности кода для случаев итерации по элементам массива с помощью указателей и с помощью индексации (§5.3.1). Если у компилятора есть разные степени оптимизации, убедитесь, влияет ли это (и как) на качество генерируемого машинного кода.

9. (*1.5) Продемонстрируйте пример, когда имеет смысл использовать объявляемое имя в инициализаторе.

10. (*1) Определите массив строк, содержащих названия месяцев. Распечатайте эти строки. Для печати передайте этот массив в функцию.

11. (*2) Читайте последовательность слов из потока ввода. Пусть слово ***Quit*** служит признаком конца ввода. Печатайте слова в порядке, в каком они были введены, но не допускайте повтора одинаковых слов. Модифицируйте программу так, чтобы перед печатью слова предварительно сортировались.

12. (*2) Напишите функцию, которая подсчитывает количество повторов пар букв в строке типа ***string***, а также в символьном массиве с терминальным нулем (строка в С-стиле). Например, пара букв "ab" входит в строку "xabaacbaxabb" дважды.

13. (*1.5) Определите структуру ***Date*** для хранения дат. Напишите функции для чтения дат из потока ввода, для вывода дат и для их инициализации.

6

Выражения и операторы

Преждевременная оптимизация —
корень всех зол.
— Д. Кнут

С другой стороны, мы не можем игнорировать
эффективность.
— Джон Бентли

Пример с калькулятором — ввод — параметры командной строки — обзор операций — побитовые логические операции — инкремент и декремент — свободная память — явное преобразование типов — сводка операторов — объявления — операторы выбора — объявления в условиях — операторы цикла — пресловутый ***goto*** — комментарии и отступы — советы — упражнения.

6.1. Калькулятор

Мы рассмотрим операторы и выражения языка C++ на примере программы калькулятора, реализующего четыре стандартных арифметических действия в форме инфиксных операций над числами с плавающей запятой. Пользователь может также определять переменные. Например, если на входе калькулятора имеется

```
r=2.5
area=pi* r* r
```

(где ***pi*** — это предопределенная переменная) то в результате работы программа-калькулятор выдаст

```
2.5
19.635
```

где ***2.5*** — это результат обработки первой строки, а ***19.635*** — второй.

Наш калькулятор состоит из четырех основных частей: синтаксического анализатора (или парсера — parser), функции ввода, таблицы символов и управляющей

программы (управляющая, ведущая программа — driver). По сути дела, эта программа является миниатюрным компилятором, где парсер выполняет синтаксический анализ, функция ввода реализует ввод исходных данных и выполняет лексический анализ, в таблице символов хранится постоянная информация, а управляющая программа осуществляет инициализацию, вывод результатов и обработку ошибок. Мы могли бы добавить к этому калькулятору массу иных возможностей, чтобы он стал более полезным (§6.6[20]), но код и без того уже достаточно велик, и кроме того, новые возможности калькулятора ничего бы нам не добавили в плане изучения языка C++.

6.1.1. Синтаксический анализатор

Вот *формальная грамматика* программы-калькулятора:

```
program:
    END                          // END это конец ввода
    expr_list END

expr_list:
    expression PRINT             // PRINT это точка с запятой
    expression PRINT expr_list

expression:                      // выражение
    expression + term
    expression - term
    term

term:
    term / primary
    term* primary
    primary

primary:                         // первичное выражение
    NUMBER                       // число
    NAME                         // имя
    NAME = expression
    -primary
    (expression)
```

Иными словами, программа есть последовательность выражений, разделенных точками с запятой. Базовыми элементами выражений служат числа, имена и операции *, / +, - (унарная и бинарная), =. Объявлять имена до их использования необязательно.

Применяемый нами стиль синтаксического анализа называется *рекурсивным спуском* (*recursive descent*). В языках типа C++, где вызов функций обходится достаточно дешево, этот стиль еще и эффективен. Каждому порождающему правилу грамматики сопоставляется своя функция, вызывающая другие функции. Терминальные символы (например, ***END***, ***NUMBER***, + и -) распознаются лексическим анализатором ***get_token***`()`; нетерминальные символы распознаются функциями синтаксического анализа, ***expr***`()`, ***term***`()` и ***prim***`()`. Как только оба операнда (под)выражения распознаны, выражение тут же вычисляется; в настоящем же компиляторе в этот момент скорее генерируется машинный код.

Входные данные для парсера поставляет функция ***get_token***(). Самый последний возврат функции ***get_token***() хранится в глобальной переменной ***curr_tok***, имеющей тип перечисления ***Token_value***:

```
enum Token_value
{
    NAME,        NUMBER,        END,
    PLUS='+',    MINUS='-',     MUL='*',    DIV='/',
    PRINT=';',   ASSIGN='=',    LP='(',     RP=')'
};

Token_value curr_tok=PRINT;
```

Представление *лексемы* (*tokens*) целочисленным кодом соответствующего символа удобно и эффективно, в том числе и при отладке программы. С этим не возникает никаких проблем, так как в известных мне системах кодировки символов нет печатных символов с однозначными кодами. Я выбрал ***PRINT*** в качестве начального значения для переменной ***curr_tok***, поскольку именно это значение она получает после вычисления выражения и вывода результата. Таким образом, система стартует в нормальном состоянии, что минимизирует потенциальные ошибки и необходимость в специальной инициализации.

Каждая функция синтаксического анализа принимает аргумент типа ***bool*** (§4.2), указывающий, должна ли функция вызвать ***get_token***() для получения очередной лексемы. Каждая такая функция вычисляет «свое выражение» и возвращает вычисленное значение. Функция ***expr***() обрабатывает сложение и вычитание. Она состоит из единственного цикла, выполняющего сложение или вычитание *термов* (по-английски *terms* — это члены алгебраических выражений, над которыми выполняются операции сложения/вычитания; сами они представлены в виде произведения/частного выражений):

```
double expr(bool get)            // сложение и вычитание
{
 double left=term(get);

 for( ; ; )                      // "вечно"
   switch(curr_tok)
   {
     case PLUS:
       left += term(true);
       break;
     case MINUS:
       left -= term(true);
       break;
     default:
       return left;
   }
}
```

Эта функция мало что делает сама по себе. В манере высокоуровневых функций из крупных программ она вызывает другие функции, которые и делают большую часть работы.

Оператор ***switch*** сравнивает значение выражения в круглых скобках с набором констант. Оператор ***break*** используется для организации принудительного выхода из оператора ***switch***. Константы, стоящие за ***case***-метками, должны отличаться друг от друга. Если значение проверяемого выражения не совпадает ни с одной из констант, то осуществляется переход на ***default***-ветвь. В общем случае, программист не обязан реализовывать ***default***-ветвь в операторе ***switch***.

Следует отметить, что выражение вроде *2 - 3 + 4* вычисляется как (*2 - 3*) + *4*, что диктуется определенной выше грамматикой.

Странное обозначение ***for***(; ;) задает *бесконечный цикл*. Это вырожденный случай оператора цикла ***for*** (§4.2); ***while***(***true***) — еще один пример на эту тему. В итоге оператор ***switch*** исполняется раз за разом, пока не встретится что-либо отличное от + или -, после чего срабатывает оператор ***return*** из ***default***-ветви.

Операции присваивания += и -= использованы для выполнения сложения или вычитания. Можно было бы писать ***left = left + term***(***true***) и ***left = left - term***(***true***) с тем же самым конечным результатом. Но выражения ***left*** += ***term***(***true***) и ***left*** -= ***term***(***true***) не только короче, но и четче фиксируют предполагающиеся к выполнению действия. Каждая операция присваивания является отдельной самостоятельной лексемой, и поэтому выражение ***a*** + = 1; ошибочно из-за лишних пробелов между + и =.

Сочетание простых операций присваивания возможно со следующими бинарными (двухоперандными) операциями

```
+    -    *    /    %    &    |    ^    <<    >>
```

так что имеются следующие *составные* (комбинированные) *операции присваивания*:

```
=   +=    -=    *=    /=    %=    &=    |=    ^=    <<=   >>=
```

Здесь % означает операцию целочисленного деления по модулю (то есть нахождение остатка); &, | и ^ — побитовые логические операции «И», «ИЛИ» и «ИСКЛЮЧАЮЩЕЕ ИЛИ»; << и >> это операции битового сдвига влево и вправо; в §6.2 дана сводная информация по всем операциям и их смыслу. Для операндов встроенных типов и любой бинарной (двухоперандной) операции @ выражение ***x*** @= ***y*** равносильно ***x*** = ***x*** @ ***y*** с той лишь разницей, что x вычисляется лишь один раз.

В главах 8 и 9 обсуждается модульная организация программ. Нашу же программу-калькулятор мы здесь упорядочиваем так, что объявления даются лишь один раз и до их использования. Единственное исключение касается функции ***expr***(), которая вызывает ***term***(), которая вызывает ***prim***(), которая вызывает ***expr***(). Этот замкнутый круг нужно как-то разорвать. Объявление

```
double expr(bool) ;
```

расположенное до определения функции ***prim***(), решает проблему.

Функция ***term***() обрабатывает умножение и деление аналогично тому, как ***expr***() обрабатывает сложение и вычитание:

```
double term (bool get)          // умножение и деление
{
  double left = prim (get) ;

  for( ; ; )
    switch (curr_tok)
```

```
    {
      case MUL :
        left *= prim(true);
        break;
      case DIV:
        if(double d = prim(true))
        {
          left /= d;
          break;
        }
        return error{"divide by 0"};
      default :
        return left;
    }
}
```

Деление на нуль не определено и обычно приводит к неприятным последствиям. Поэтому мы проверяем делитель на нуль до выполнения операции деления, и вызываем функцию ***error***() в случае нулевого значения делителя. Функция ***error***() описывается в §6.1.4.

Переменная ***d*** определяется в программе ровно там, где она требуется, и тут же при этом инициализируется. Областью видимости переменной, определенной в условии, является тело условной конструкции, а значение этой переменной совпадает с величиной условного выражения (§6.3.2.1). Поэтому выражение ***left*** /= ***d*** вычисляется тогда и только тогда, когда ***d*** не равно нулю.

Функция ***prim***(), обрабатывающая первичные элементы, похожа на ***expr***() и ***term***(), но поскольку тут мы забрались ниже по иерархии вызовов некоторая реальная работа уже выполняется (и цикл не нужен):

```
double number_value;
string string_value;

double prim(bool get)              // обработка первичных выражений
{
  if(get) get_token();

  switch(curr_tok)
  {
    case NUMBER:
    {
      double v = number_value;
      get_token();
      return v;
    }
    case NAME:
    {
      double& v = table[string_value];
      if(get_token() == ASSIGN) v = expr(true);
      return v;
    }
    case MINUS:                    // унарный минус
      return -prim(true);
```

```
    case LP:
    {
      double e = expr(true);
      if(curr_tok != RP) return error("')' expected");
      get_token();                // пропустить ')'
      return e;
    }
    default:
      return error("primary expected");
  }
}
```

Когда встречается *NUMBER* (то есть целый литерал или литерал с плавающей запятой), возвращается его значение. Вводимое функцией ***get_token***() значение помещается в глобальную переменную ***number_value***. Обычно глобальная переменная в программе сигнализирует о недостаточно четкой ее структуре — например, проведена некоторая оптимизация. Таков и наш случай. В идеале лексема состоит из двух частей: значения, определяющего тип лексемы (***Token_value*** в данной программе), и величины самой лексемы (где это требуется). У нас же имеется единственная переменная ***curr_tok***, из-за чего и потребовалась глобальная переменная ***number_vaule*** для хранения последнего прочитанного *NUMBER*. Задача исключить эту глобальную переменную предложена ниже в качестве самостоятельного упражнения (§6.6[21]). В действительности, нет жесткой необходимости сохранять значение ***number_value*** в локальной переменной ***v*** перед вызовом ***get_token***(). Для каждого допустимого ввода наш калькулятор всегда выполняет вычисления с единственным числом, после чего читает новое. Но в случае ошибки сохраненное число и его вывод помогут пользователю.

Аналогично тому, как последнее значение *NUMBER* хранится в переменной ***number_value***, строковое представление последнего *NAME* хранится в ***string_value***. Прежде, чем обработать имя, калькулятору нужно заглянуть вперед и узнать, как это имя используется — читается или оно участвует в операции присваивания. В любом случае происходит обращение к *таблице символов*, имеющей тип ***map*** (§3.7.4, §17.4.1):

```
map<string, double> table;
```

Когда таблица символов ***table*** индексируется строкой, возвращается значение типа ***double***, ассоциированное с указанной строкой. Например, если пользователь вводит

```
radius=6378.388;
```

то калькулятор выполняет следующее:

```
double& v = table["radius"];
// ... expr() вычисляет значение, которое будет присвоено ...
v = 6378.388;
```

Ссылка ***v*** настраивается на числовое значение типа double, ассоциированное с именем ***radius***, и сохраняет к нему доступ, пока функция ***expr***() не выделит число ***6378.388*** из входного потока символов.

6.1.2. Функция ввода

Очень часто считывание ввода является самой запутанной частью программы. Это объясняется тем, что программе приходится взаимодействовать с человеком — учитывать его причуды, пристрастия и просто случайные ошибки. Попытка заставить человека действовать как машина часто воспринимается им (вполне справедливо) как оскорбление. Задача процедуры низкоуровневого ввода состоит в чтении потока символов и выделения из него более высокоуровневых лексем. Эти лексемы затем подаются на вход еще более высокоуровневым процедурам. В нашей программе низкоуровневый ввод обрабатывается функцией *get_token*(). По счастью, написание процедур низкоуровневого ввода не является каждодневной задачей, ибо во многих системах для этого имеются готовые стандартные процедуры.

Я выбрал двухэтапную тактику построения функции *get_token*(). Сначала я напишу обманчиво простую версию, которая все сложности перекладывает на пользователя программы. Затем я модифицирую ее в окончательный вариант, который выглядит менее элегантно, зато прост в использовании.

Общий план состоит в считывании символа, определении по этому символу типа лексемы и возврате соответствующего *Token_value*.

Начальные операторы функции считывают первый непробельный символ в переменную *ch* и проверяют успешность ввода:

```
Token_value get_token ()
{
  char ch = 0;
  cin>>ch;

  switch (ch)
  {
    case 0:
      return curr_tok=END;      // присваивание и возврат
```

По умолчанию, операция >> пропускает пробельные символы (то есть собственно пробел, табуляцию, переход на новую строку и т.д.) и не изменяет значение переменной *ch* в случае неудачи ввода. Отсюда следует, что *ch* == *0* означает конец ввода.

Операция присваивания вырабатывает значение, совпадающее с новой величиной переменной, стоящей в левой части операции присваивания. Поэтому я присваиваю *END* переменной *curr_tok* и возвращаю это же значение в рамках единственного оператора. Это улучшает надежность сопровождения программы, так как в случае двух операторов программист может изменить лишь один из них, забыв про другой.

Теперь взглянем на отдельные ветви оператора *switch* до того, как будет представлен весь текст функции. Для символа окончания выражений, то есть символа ';', круглых скобок и символов операций, выполняемых калькулятором, возвращаются их числовые коды:

```
    case ';':
    case '*':
    case '/':
    case '+':
    case '-':
```

```
case '(' :
case ')' :
case '=' :
  return curr_tok=Token_value(ch) ;
```

Для чисел поступаем следующим образом:

```
case '0' : case '1' : case '2' : case '3' : case '4' :
case '5' : case '6' : case '7' : case '8' : case '9' :
case '.' :
  cin.putback(ch) ;
  cin>>number_value;
  return curr_tok=NUMBER;
```

Может быть и не слишком хорошо с точки зрения восприятия располагать case-ветви оператора ***switch*** горизонтально, но уж больно утомительно писать однообразные строки одну под другой. Суть же кода очень проста, так как операция >> читать константы с плавающей запятой в переменные типа ***double*** умеет изначально. Выявив характерный для чисел начальный символ (цифра или точка), возвращают его в поток ***cin***, после чего вся константа читается в переменную ***number_value***.

Аналогично работаем с именами:

```
default:                    // NAME, NAME =, или ошибка
  if(isalpha(ch))
  {
    cin.putback(ch) ;
    cin>>string_value;
    return curr_tok=NAME;
  }
  error("bad token") ;
  return curr_tok=PRINT;
```

Чтобы избежать перечисления всех алфавитных символов в ***case***-ветвях, используем *стандартную библиотечную функцию* ***isalpha***() (§20.4.2). Операция >> читает символы в строку (в нашем случае, в переменную ***string_value***) до тех пор, пока не встретится пробельный символ. Это значит, что во входном тексте пользователь программы должен после имени (и до знака операции) располагать пробельные символы. Это крайне неудобно и мы еще вернемся к этому вопросу в §6.1.3.

И вот, наконец, полный текст функции ввода:

```
Token_value get_token()
{
  char ch=0;
  cin>>ch;

  switch(ch)
  {
    case 0:
      return curr_tok=END;
    case ';' :
    case '*' :
    case '/' :
```

```
        case '+':
        case '-':
        case '(':
        case ')':
        case '=':
          return curr_tok=Token_value(ch);
        case '0': case '1': case '2': case '3': case '4':
        case '5': case '6': case '7': case '8': case '9':
        case '.':
          cin.putback(ch);
          cin>>number_value;
          return curr_tok=NUMBER;
         default:                        // NAME, NAME =, или ошибка
          if(isalpha(ch))
          {
            cin.putback(ch);
            cin>>string_value;
            return curr_tok=NAME;
          }
          error("bad token");
          return curr_tok=PRINT;
      }
    }
```

Преобразование операций к типу ***Token_value*** имеет тривиальный смысл, так как элементы этого перечисления, соответствующие операциям, имеют значения, совпадающие с кодировками символов операций (см. также §4.8).

6.1.3. Низкоуровневый ввод

Работа с текущей версией калькулятора имеет ряд неудобств. Например, нужно не забывать про точку с запятой, иначе калькулятор не выведет вычисленного значения. Необходимость же в обязательном порядке проставлять пробелы после имен, вообще просто нонсенс. Действительно, в данной версии калькулятора ***x=7*** трактуется как идентификатор, а вовсе не набор из идентификатора ***x***, операции = и числа *7*. Все перечисленные неудобства устраняются путем замены в теле функции ***get_token***() простых стандартных операций типоориентированного ввода на операции посимвольного ввода.

Начнем с того, что предложим наряду с точкой с запятой использовать и перевод строки для индикации конца выражений:

```
Token_value get_token()
{
  char ch;

  do                          // пропустить все пробельные символы кроме '\n'
  {
    if(!cin.get(ch)) return curr_tok = END;
  }while(ch!='\n' && isspace(ch));

  switch(ch)
  {
```

```
case ';':
case '\n':
  return curr_tok=PRINT;
```

Здесь использован *оператор цикла* **do**, который эквивалентен циклическому оператору while, за исключением того, что *его тело всегда исполняется хотя бы один раз.* Вызов **cin.get(ch)** читает один символ из стандартного потока ввода в переменную **ch**. *По умолчанию,* **get()** *не выполняет автоматически пропуск пробельных символов* так, как это делает операция >>. Проверка **if(!cin.get(ch))** выявляет невозможность чтения символа из cin; в этом случае возвращается **END** и работа с калькулятором завершается. Операцию ! (логическое отрицание) приходится использовать из-за того, что **get()** возвращает **true** в случае успешного чтения.

Стандартная библиотечная функция **isspace()** проверяет, является ли символ *пробельным* (§20.4.2); **isspace(c)** возвращает ненулевое значение для пробельных символов и нуль в противном случае. Функция **isspace()** выполняется быстрее, чем наш альтернативный перебор всех пробельных символов. Имеются также стандартные функции для выявления цифр — **isdigit()** — букв — **isalpha()**, а также цифр или букв — **isalnum()**.

После пропуска пробельных символов следующий за ними символ используется для выявления типа лексемы.

Проблема, вызванная чтением строки операцией >> вплоть до пробельных символов, преодолевается последовательным посимвольным чтением до тех пор, пока не встретится символ, отличный от буквы или цифры:

```
default:                    // NAME, NAME=, или ошибка
  if(isalpha(ch))
  {
    string_value=ch;
    while(cin.get(ch) && isalnum(ch)) string_value.push_back(ch);
    cin.putback(ch);
    return curr_tok=NAME;
  }
  error("bad token");
  return curr_tok=PRINT;
```

Оба рассмотренных улучшения достигаются модификацией четко ограниченных небольших участков кода. Конструирование программ таким образом, что их последующие модификации могут ограничиваться локальными изменениями кода, является важной целью проектирования.

6.1.4. Обработка ошибок

Из-за того, что наша программа чрезвычайно проста, обработка ошибок для нее не является предметом первостепенного интереса. Наша функция обработки ошибок просто подсчитывает их количество, выводит сообщение и завершает работу:

```
int no_of_errors;

double error(const string& s)
{
  no_of_errors++;
```

```
    cerr<< "error: " << s << '\n';
    return 1;
}
```

Поток **cerr** — это *небуферизованный выходной поток*, который обычно используется *для вывода сообщений об ошибках* (§21.2.1).

Наша функция возвращает значение по той причине, что обычно ошибки возникают в процессе вычисления выражений, так что следует либо полностью прервать этот процесс, либо вернуть некоторое значение, которое не вызовет осложнений в дальнейшей работе. Для нашего простого калькулятора годится второй подход. Если бы **get_token**() отслеживала номер строки исходного текста, функции **error**() имело бы смысл информировать пользователя о том, где примерно произошла ошибка. Это было бы особенно полезно в случае неинтерактивного использования программы-калькулятора (§6.6[19]).

Часто нужно завершать работу программы при возникновении ошибок, ибо нет разумного продолжения их работы. Это можно сделать вызовом *библиотечной функции* **exit**(), которая *сначала освобождает ресурсы* (вроде потоков вывода), и *только после этого завершает программу*, возвращая значение, совпадающее с аргументом ее вызова (§9.4.1.1).

Более регулярный метод обработки ошибок основан на работе с исключениями (§8.3, глава 14), но для программы из 150 строк вполне достаточно уже сделанного нами.

6.1.5. Управляющая программа

Теперь нам недостает лишь управляющей программы, которая запускает и оркеструет всю работу. В нашем случае для этого достаточно одной функции **main**():

```
int main ()
{
    table["pi"]=3.1415926535897932385;         // вводим предопределенные имена
    table["e"]=2.7182818284590452354;

    while(cin)
    {
        get_token();
        if(curr_tok==END) break;
        if(curr_tok==PRINT) continue;
        cout<< expr(false) << '\n';
    }

    return no_of_errors;
}
```

По устоявшемуся соглашению функция **main**() возвращает нуль в случае нормального завершения и ненулевое значение в противном случае (§3.2). Возврат количества ошибок отлично развивает эту идею. Вся инициализация свелась у нас к формированию таблицы предопределенных имен.

В теле функции **main**() основная работа сосредоточена в цикле, где считывается выражение и выводится результат его вычисления. Последнее выполняется следующей строкой кода:

```
cout<<expr(false)<<'\n';
```

Аргумент ***false*** сообщает функции ***expr***(), что ей не надо вызывать ***get_token***() для выделения лексемы.

Проверка ***cin*** для каждой итерации цикла гарантирует корректное завершение программы, если с входным потоком возникнут какие-нибудь проблемы, а сравнение с ***END*** гарантирует корректное завершение цикла, если ***get_token***() обнаружит *конец ввода*[1]. Оператор ***break*** осуществляет принудительный выход из ближайшего оператора ***switch*** или цикла. Сравнение с ***PRINT*** (то есть с '\n' и ';') освобождает функцию ***expr***() от ответственности за обработку пустых выражений. Оператор *continue* означает переход к следующей итерации цикла (по-другому можно сказать, что в самый конец тела цикла), так что код

```
while (cin)
{
  //...
  if (curr_took==PRINT) continue;
  cout<< expr(false) << '\n';
}
```

эквивалентен

```
while (cin)
{
  //...
  if (curr_took!=PRINT)
    cout<< expr(false) << '\n';
}
```

6.1.6. Заголовочные файлы

Наша программа-калькулятор использует средства стандартной библиотеки, из-за чего в окончательный код нужно включить следующие директивы ***#include*** препроцессора:

```
#include <iostream>        // I/O (ввод/вывод)
#include <string>          // string (строки)
#include <map>             // map (ассоциативные массивы)
#include <cctype>          // isalpha(), и т.д.
```

Все средства из перечисленных здесь заголовочных файлов определяются в пространстве имен ***std***, так что имена из этих файлов нужно либо полностью квалифицировать префиксом ***std***::, либо заранее внести их в глобальное пространство имен следующим образом:

```
using namespace std;
```

Я выбрал последнее, чтобы не смешивать в одну кучу обсуждение вопросов обработки выражений и концепций модульного построения программ. В главах 8 и 9 обсуждаются способы модульной организации программы-калькулятора с приме-

[1] Это служебный символ end-of-file — он генерируется клавиатурой в ответ на нажатие специальной комбинации клавиш, чаще всего — Ctrl+Z. — *Прим. ред.*

нением сегментирующих средств в виде пространств имен и набора исходных файлов. На многих системах помимо указанных выше заголовочных файлов имеются еще и их эквиваленты с расширением *.h*, и в которых классы, функции и т.д. определяются в глобальном пространстве имен (§9.2.1, §9.2.4, §B.3.1).

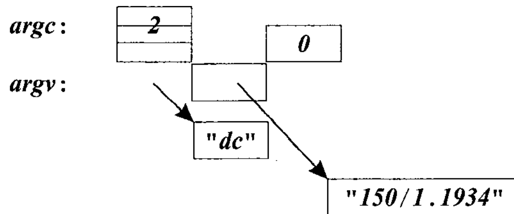
6.1.7. Аргументы командной строки

После того как я написал и оттестировал программу-калькулятор, я обнаружил, что пользоваться ею не всегда удобно: нужно ее запустить, затем ввести выражение, и в конце концов, завершить работу программы. Чаще всего я использовал калькулятор для вычисления единственного выражения. Если бы это выражение можно было представить параметром командной строки, то удалось бы сэкономить несколько нажатий клавиш.

Программа начинает свою работу с функции *main*() (§3.2, §9.4). Ей передаются два аргумента, первый из которых задает общее количество аргументов командной строки (обычно обозначается argc), а второй — это массив аргументов (обычно обозначаемый *argv*). Эти аргументы передаются функции *main*() оболочкой, ответственной за запуск программы; если оболочка не может передать аргументы, то *argc* устанавливается в нуль. Так как соглашения о вызове функции *main*() взяты из языка С, то используется массив строк в стиле С (*char** *argv*[*argc*+*1*]). Имя программы (как оно записано в командной строке) передается с помощью *argv*[*0*]. Список аргументов имеет завершающий нуль, то есть *argv*[*argc*] == *0*. Например, для командной строки

dc 150/1.1934

ее аргументы имеют следующие значения:



Получить доступ к аргументам командной строки несложно. Вопрос в том, как их использовать с минимальными изменениями в готовой программе. Для этого предлагается читать из командной строки так же, как мы читали входные данные из входного потока. Неудивительно, что *поток для чтения из строки называется* ***istringstream***. К сожалению, отсутствует способ заставить *cin* ссылаться на поток *istringstream*. Поэтому мы должны заставить функции ввода нашей программы-калькулятора работать с *istringstream*. Более того, мы должны их заставить работать либо с *istringstream*, либо с cin, в зависимости от конкретного вида командной строки.

Простейшим решением будет определить глобальный указатель *input* и настроить его на нужный поток, а также заставить все процедуры ввода использовать его:

```
istream* input ;                    // указатель на поток ввода

int main (int argc, char* argv[ ])
{
  switch (argc)
  {
    case 1:
      input= &cin;
      break;
    case 2:
      input=new istringstream (argv[1]) ;
      break;
    default:
      error ("too many arguments") ;
      return 1;
  }

  table["pi"]=3.1415926535897932385;
  table["e"]=2.7182818284590452354;

  while (*input)
  {
    get_token () ;
    if(curr_tok==END) break;
    if(curr_tok==PRINT) continue;
    cout<<expr(false)<<'\n' ;
  }

  if(input!=&cin) delete input;

  return no_of_errors;
}
```

Поток ***istringstream*** является *потоком ввода*, который *читает из строки, переданной в качестве аргумента* (§21.5.3). По достижению конца читаемой строки поведение потока ***istringstream*** точно такое же, как у остальных потоков ввода (§3.6, §21.3.3). Чтобы использовать в программе ***istringstream***, нужно директивой ***#include*** препроцессора включить заголовочный файл ***<sstream>***.

Было бы совсем нетрудно заставить функцию ***main*** () читать большее число аргументов командной строки, но это представляется ненужным, так как несколько выражений можно передать одним аргументом:

```
dc "rate=1.1934;150/rate;19.75/rate;217/rate"
```

Я применил двойные кавычки, поскольку ; (точка с запятой) на моей UNIX системе служит разделителем команд. Другие операционные системы имеют свои соглашения по передаче аргументов программам при их запуске.

Нельзя назвать наше решение элегантным, поскольку все процедуры ввода приходится переделывать под использование ****input*** вместо ***cin***, с тем чтобы они могли читать из разных источников. Этих изменений можно было избежать, если бы я заранее предвидел эту ситуацию и применил что-нибудь вроде ***input*** с самого начала. Более общий и полезный подход состит в том, что источник ввода должен быть параметром для модуля калькулятора. Фундаментальная проблема данной программы-калькулятора состоит в том, что так называемый «калькулятор» реализуется

в ней просто *набором функций и данных*. Нет ни *модуля* (§2.4), ни *объекта* (§2.5.2), *представляющих калькулятор в явном виде*. Если бы я разрабатывал модуль калькулятора или тип калькулятора, то, естественно, задумался бы над его параметрами (§8.5[3], §10.6[16]).

6.1.8. Замечания о стиле

Программистам, не знакомым с ассоциативными массивами, применение библиотечного типа ***map*** для представления таблицы символов может показаться нарочитым высокоуровневым трюкачеством. Это не так. И стандартная, и другие библиотеки для того и создаются, чтобы ими пользовались. Часто библиотеки разрабатываются и реализуются с такой тщательностью, какую отдельный программист не может себе позволить в процессе написания участка кода, применимого лишь в одной программе.

Взглянув на нашу программу-калькулятор, особенно на ее первую версию, можно заметить, что в ней мало традиционно низкоуровневого кода в С-стиле. Необходимость низкоуровневых трюков была устранена за счет применения библиотечных классов ***ostream***, ***string*** и ***map*** (§3.4, §3.5, §3.7.4, глава 17).

Обратите внимание, как редко встречаются арифметические операции, циклы и даже присваивания. Так и должно обстоять дело в программах, не управляющих непосредственно аппаратурой и не реализующих низкоуровневых абстракций.

6.2. Обзор операций языка C++

В данном разделе дается обзор выражений и операций языка C++, а также примеры на эту тему. Каждая операция сопровождается названием и примером выражения, использующего эту операцию. Ниже в таблице операций *class_name* означает имя класса; *member* — имя члена класса; *object* — выражение, дающее объект класса; *pointer* — выражение, дающее указатель; *expr* — выражение; *lvalue* — выражение, обозначающее неконстантный объект; *type* — имя типа в самом общем смысле (вместе с `*`, `()` и т.д.), если помещено в круглые скобки, а иначе есть ограничения (§A.5).

Синтаксис выражений не зависит от типа операндов. Однако смысл операций, указанный ниже, имеет место лишь для встроенных типов (§4.1.1). Смысл операций над пользовательскими типами данных можно задавать самостоятельно (§2.5.2, глава 11).

Операции	
разрешение области видимости	*class-name* :: *member*
разрешение области видимости	*namespace-name* :: *member*
глобально	:: *name*
глобально	:: *qualified-name*
выбор члена	*object* . *member*
выбор члена	*pointer* -> *member*
доступ по индексу	*pointer* [*expr*]
вызов функции	*expr* (*expr-list*)

Операции	
конструирование значения	*type* (*expr-list*)
постфиксный инкремент	*lvalue* ++
постфиксный декремент	*lvalue* --
идентификация типа	***typeid*** (*type*)
идентификация типа во время выполнения	***typeid*** (*expr*)
преобразование с проверкой во время выполнения	***dynamic_cast*** <*type*> (*expr*)
преобразование с проверкой во время компиляции	***static_cast*** <*type*> (*expr*)
преобразование без проверки	***reinterpret_cast***<*type*> (*expr*)
константное преобразование	***const_cast***<*type*> (*expr*)
размер объекта	***sizeof*** *expr*
размер типа	***sizeof*** (*type*)
префиксный инкремент	++ *lvalue*
префиксный декремент	-- *lvalue*
дополнение	~ *lvalue*
отрицание	! *expr*
унарный минус	- *expr*
унарный плюс	+ *expr*
адрес	& *lvalue*
разыменование	* *expr*
создать (выделить память)	***new*** *type*
создать (выделить память и инициализировать)	***new*** *type* (*expr-list*)
создать (разместить)	***new*** (*expr-list*) *type*
создать (разместить и инициализировать)	***new*** (*expr-list*) *type* (*expr-list*)
уничтожить (освободить память)	***delete*** *pointer*
уничтожить массив	***delete*** [ ] *pointer*
приведение (преобразование типа)	(*type*) *expr*
выбор члена	*object* .* *pointer-to-member*
выбор члена	*pointer* ->* *pointer-to-member*
умножение	*expr* * *expr*
деление	*expr* / *expr*
остаток от деления (деление по модулю)	*expr* % *expr*
сложение (плюс)	*expr* + *expr*
вычитание (минус)	*expr* - *expr*
сдвиг влево	*expr* << *expr*
сдвиг вправо	*expr* >>*expr*
меньше	*expr* < *expr*
меньше или равно	*expr* <= *expr*
больше	*expr* > *expr*
больше или равно	*expr* >= *expr*
равно	*expr* == *expr*
не равно	*expr* != *expr*
побитовое И (AND)	*expr* & *expr*
побитовое исключающее ИЛИ (OR)	*expr* ^ *expr*
побитовое ИЛИ (OR)	*expr* | *expr*
логическое И (AND)	*expr* && *expr*

Операции	
логическое ИЛИ (OR)	*expr* || *expr*
условное выражение	*expr* ? *expr*: *expr*
простое присваивание умножение и присваивание деление и присваивание остаток и присваивание сложение и присваивание вычитание и присваивание сдвиг влево и присваивание сдвиг вправо и присваивание И и присваивание ИЛИ и присваивание исключающее ИЛИ и присваивание	*lvalue* = *expr* *lvalue* *= *expr* *lvalue* /= *expr* *lvalue* %= *expr* *lvalue* += *expr* *lvalue* -= *expr* *lvalue* "= *expr* *lvalue* "= *expr* *lvalue* &= *expr* *lvalue* |= *expr* *lvalue* ^= *expr*
генерация исключения	***throw*** *expr*
запятая (следование)	*expr* , *expr*

В каждой секции данной таблицы сосредоточены операции с одинаковым *приоритетом* (*precedence*). Операции из вышестоящих секций имеют приоритет, более высокий по отношению к операциям, расположенным в нижележащих секциях. Например, ***a+b*c*** означает именно ***a+(b*c)***, а не ***(a+b)*c***, так как операция * имеет более высокий приоритет, чем операция +. Аналогично, выражение ****p++*** означает ****(p++)***, а не ***(*p)++***.

Префиксные операции и операции присваивания *правоассоциативны* (*right-associative*); все остальные операции — *левоассоциативны* (*left-associative*). Например, ***a=b=c*** означает ***a=(b=c)***, в то время как ***a+b+c*** есть ***(a+b)+c***.

Несколько грамматических правил невозможно выразить в терминах приоритета (называемого, также, силой связывания) и ассоциативности. Например, ***a=b<c?d=e:f=g*** означает ***a=((b<c)?(d=e):(f=g))***, но для подтверждения этого факта нужно обратиться к сводке грамматических правил (§A.5).

Грамматические правила применяются после того, как из отдельных символов построены лексемы (§A.3), причем лексемы строятся из потенциально наиболее длинных последовательностей. Например, && означает единственную операцию, а не две операции &; выражение ***a+++1*** означает ***(a++)+1***.

6.2.1. Результаты операций

Результирующий тип арифметических операций определяется набором правил, известных как «*стандартные арифметические преобразования*» (§C.6.3). Их общая цель сводится к тому, чтобы получить тип «наибольшего» операнда. Например, если у бинарной операции один из операндов имеет тип с плавающей запятой, то вычисления выполняются в арифметике чисел с плавающей запятой и, соответственно, таков и тип результата этой операции. Если у бинарной операции один из операндов имеет тип ***long***, то вычисления выполняются в арифметике длинных целых, и возвращается результат типа ***long***. Операнды «меньших» чем ***int*** типов (такие как ***bool*** или ***char***) преобразуются к ***int*** перед выполнением операции.

Операции сравнения ==, <= и т.д. *вырабатывают результат логического типа.* Смысл и тип результатов операций, определяемых пользователем, вытекают из их объявления (§11.2).

Если нет логических препятствий, то результат операции с операндом ***lvalue*** совпадает с ***lvalue***, которое операция вырабатывает для этого операнда. Например:

```
void f(int x, int y)
{
  int j=x=y;              // значением x=y является значение x после присваивания
  int* p=&++x;            // p указывает на x
  int* q=&(x++);          // error: x++ не есть lvalue (это не значение, хранимое в x)
  int* pp=&(x>y?x:y);     // адрес целого с большим значением
}
```

Если второй и третий операнды операции ?: являются lvalue одного и того же типа, то результат операции есть lvalue того же самого типа. В целом, такой «сохраняющий» стиль обращения с lvalue обеспечивает чрезвычайную гибкость операций. Особую важность это приобретает в случаях, когда требуется писать код, одинаково подходящий и эффективный как для встроенных, так и для пользовательских типов данных (например, при написании шаблонов или программ, генерирующих код C++).

Результат операции ***sizeof*** есть *беззнаковый интегральный тип* ***size_t***, определенный в заголовочном файле <***cstddef***>. Типом *разности указателей* служит *знаковый интегральный тип* ***ptrdiff_t***, также определенный в файле <***cstddef***>.

Конкретные реализации C++ не обязаны контролировать арифметическое переполнение и вряд ли какая из них делает это. Например:

```
void f()
{
  int i=1;
  while(0<i) i++;
  cout<< "i has become negative!" << i << '\n';
}
```

Эта функция попытается (в итоге) увеличить на единицу максимально возможное целое значение. Результат не определен, но в типичных случаях произойдет перескок к отрицательным значением (на моей машине ***-2147483648***). Также не определен результат деления на нуль, но обычно это заканчивается принудительным остановом программы. Стандартные исключения (§14.10) для деления на нуль и переполнений не генерируются.

6.2.2. Последовательность вычислений

Порядок вычисления подвыражений в рамках объемлющих выражений не определен. В часности, нельзя рассчитывать на то, что выражения вычисляются слева направо. Например:

```
int x=f(2)+g(3);      // неизвестно, что будет вызвано первым - f() или g()
```

При отсутствии ограничений на порядок вычисления выражений можно генерировать более эффективный машинный код. Это же, однако, может приводить к неопределенным результатам. Например,

```
int i=1;
v[i]=i++;            // результат не определен
```

может вычисляться как *v[1]=1*, или как *v[2]=1*, или даже еще более странным образом. Компиляторы могут предупреждать о подобных *неоднозначностях* (*ambiguities*), а могут, к сожалению, и не предупреждать.

Для операции `,` (*операция запятая — comma operator*), а также для операций `&&` (*логическое «И»*) и `||` (*логическое «ИЛИ»*) гарантируется, что их *левый операнд будет вычислен раньше, чем правый операнд*. Например, с помощью выражения *b=(a=2,a+1)* переменной *b* будет присвоено значение *3*. Примеры использования операций `||` и `&&` даны в §6.2.3. Для встроенных типов правый операнд операции `&&` вычисляется лишь в случае, когда левый операнд равен ***true***. Для операции `||` правый операнд вычисляется только в случае, когда левый операнд равен ***false***. Эти правила часто называют *редуцированной схемой вычислений* (*short-circuit evaluation*). Обратите внимание на то, что *запятая как операция следования* (*sequencing operator*) логически отличается от запятой, используемой в качестве разделителя аргументов функций. К примеру, в следующем коде

```
f1(v[i],i++);        // два аргумента
f2((v[i],i++));      // один аргумент
```

вызов функции *f1()* оперирует двумя аргументами, *v[i]* и *i++*, и порядок вычисления аргументов не определен. *Нельзя полагаться на определенный порядок вычисления аргументов вызова функций* — это плохой стиль программирования, приводящий к непредсказуемому поведению программы. Вызов функции *f2()* оперирует единственным аргументом — выражением с операцией следования (операцией запятая), чье значение равно *i++*.

Принудительную группировку выражений можно осуществить с помощью круглых скобок. Например, *a*b/c* означает *(a*b)/c*, но применение круглых скобок позволяет навязать иной порядок вычислений исходного выражения — *a*(b/c)*. Для вычислений с плавающей запятой выражения *(a*b)/c* и *a*(b/c)* могут давать существенно разные результаты, так что компилятор обязан соблюдать указанный порядок вычисления выражений.

6.2.3. Приоритет операций

Уровни приоритетов операций и правила ассоциативности отражают естественные способы их использования. Например:

```
if(i<=0 || max<i)        // ...
```

означает «если *i* меньше или равно *0* или если *max* меньше *i*». То есть это эквивалентно следующему выражению

```
if((i<=0) || (max<i))    // ...
```

а не бессмысленному (хотя и формально корректному) выражению

```
if(i <= (0||max) < i)    // ...
```

Круглые скобки можно использовать всегда, когда у программиста есть сомнение. Чем сложнее выражения, тем чаще применяют круглые скобки. В то же время, слишком сложные выражения могут приводить к ошибкам. Поэтому, если вы чув-

ствуете настоятельную необходимость в круглых скобках, то возможно стоит применить вспомогательные (промежуточные) переменные и избавиться от сложных выражений.

Бывают случаи, когда приоритет операций не обеспечивает очевидности порядка вычислений. Например:

```
if(i&mask==0)        //...
```

Наложение маски (переменная mask) на *i* и сравнение результата с нулем является хотя и очевидной, но неправильной интерпретацией. Из-за того, что у операции == приоритет выше, чем у операции &, выражение на самом деле эквивалентно *i*&(*mask*==*0*). К счастью, в большинстве подобных случаев компилятор выдает предупреждение о возможной ошибке. В общем, нужно применить группирующие круглые скобки:

```
if((i&mask)==0)      //...
```

Следующее выражение работает не так, как ожидал бы математик:

```
if(0<=x<=99)         //...
```

Это формально корректное выражение интерпретируется как (*0*<=*x*)<=*99*, где первое сравнение вырабатывает ***true*** или ***false***. Эти логические значения неявно преобразуются в *1* или *0*, а затем сравниваются с *99*, что всегда дает ***true***. Для проверки попадания значения *x* в диапазон *0..99* можно написать

```
if(0<=x && x<=99)    //...
```

Для новичков типично ошибочное применение операции = (операция присваивания) вместо операции == (операция сравнения на равенство):

```
if(a=7)              //...
```

Это довольно естественно, поскольку во многих языках программирования знак = означает «равно». Но к счастью, большинство компиляторов распознают подобного рода проблемы и предупреждают программиста о возможных ошибках.

6.2.4. Побитовые логические операции

Побитовые логические операции (*bitwise logical operators*) &, |, ^, ~, >> и << применяются к интегральным типам и перечислениям, то есть к типам ***bool***, ***char***, ***short***, ***int***, ***long*** (возможно, с модификатором ***unsigned***). Обычные арифметические преобразования определяют тип результата (§C.6.3).

Типичным примером использования побитовых логических операций служит задача построения небольшого множества (битового вектора). Каждому элементу множества соответствует один бит беззнакового целого, так что количество элементов множества ограничивается количеством бит. Бинарная операция & трактуется как операция пересечения множеств, операция | трактуется как объединение множеств, операция ^ — как симметричная разность, и операция ~ — как дополнение. Для именования элементов такого множества можно использовать перечисление. Вот маленький пример, взятый из реализации потоков типа ***ostream***:

```
enum ios_base::iostate
{
 goodbit=0, eofbit=1, failbit=2, badbit=4
};
```

Можно устанавливать и проверять состояние потока следующим образом:

```
state=goodbit;
// ...
if(state&(badbit|failbit))          // с потоком проблемы
```

Здесь дополнительные круглые скобки необходимы, так как приоритет операции `&` выше, чем приоритет операции `|`.

Достигнув конца ввода, функция может просигнализировать об этом следующим образом:

```
state|=eofbit;
```

где с помощью операции `|=` к текущему состоянию потока добавляется ***eofbit***, в то время как простое присваивание ***state=eofbit*** обнулило бы все остальные биты.

Флаги состояния потока доступны и клиентам (то есть вне реализации потоков). Например, клиент может посмотреть, как различаются состояния двух потоков:

```
int diff=cin.rdstate()^cout.rdstate();     // rdstate() возвращает состояние
```

Вычисление различий в состояниях потоков встречается не столь уж и часто. Но для других типов, базирующихся на битовых векторах, эта задача весьма актуальна. Например, важно сравнивать битовые вектора, один из которых представляет собой набор обрабатываемых прерываний, а другой — набор прерываний, ожидающих обработки.

Обратите внимание на то, что вся эта возня с битами выполняется внутри реализации потоков, а не в клиентском коде. Удобная манипуляция битами может быть и важна, но из соображений надежности, сопровождаемости, переносимости и т.д. ее лучше запрятать поглубже, на нижние уровни реализации системы. За более подробными сведениями о множествах обратитесь к описаниям таких типов стандартной библиотеки, как ***set*** (§17.4.3), ***bitset*** (§17.5.3) и ***vector<bool>*** (§16.3.11).

Удобно использовать битовые поля (§С.8.1), чтобы выполнять битовые сдвиги и маскирование с целью извлечения отдельных бит из целочисленного слова. Это, конечно, можно сделать и с помощью побитовых логических операций. Например, можно следующим образом извлечь средние 16 бит из 32-разрядного ***long***:

```
unsigned short middle(long a) {return(a>>8)&0xffff;}
```

Не путайте побитовые логические операции с логическими операциями `&&`, `||` и `!`. Последние возвращают ***true*** или ***false*** и в основном используются в условных выражениях операторов ***if***, ***while*** или ***for*** (§6.3.2, §6.3.3). Например, `!`***0*** (не нуль) есть ***true***, в то время как `~`***0*** (битовое «НЕ») дает набор всех битов, равных единице, что в двоичном дополнительном коде означает −***1***.

6.2.5. Инкремент и декремент

Операция `++` выражает *инкремент* (*увеличение на единицу*) самым непосредственным образом, в то время как применение комбинации из сложения и присваи-

вания делает то же самое лишь косвенно. По определению, *++lvalue* означает *lvalue+=1*, что в свою очередь означает *lvalue=lvalue+1*, если у *lvalue* нет побочных эффектов. Выражение, возвращающее подлежащий инкременту объект, вычисляется один (и только один) раз. Операция декремента записывается как --. Операции ++ и -- *используются как в префиксной, так и в постфиксной формах*. Значением выражения *++x* является новое (инкрементированное) значение *x*. Например, *y=++x* эквивалентно *y=(x+=1)*. Значением же выражения *x++* является старое значение *x*. Например, *y=x++* эквивалентно *y=(t=x, x+=1, t)*, где переменная *t* имеет тот же тип, что и *x*.

Как и в случае сложения и вычитания указателей, *операции ++ и -- работают над указателями в единицах размера указуемых объектов*, что удобно для массивов; *p++* заставляет указатель *p* настроиться на следующий элемент массива (§5.3.1).

Операции ++ и -- особенно полезны при инкрементировании и декрементировании переменных в циклах. Например, копирование строк с терминальным нулем можно выполнить следующим образом:

```
void cpy(char* p, const char* q)
{
  while(*p++=*q++) ;
}
```

Язык C++ (как и язык C) либо сильно любят, либо сильно ненавидят за возможность написания *кода, ориентированного на выражения* (*expression-oriented coding*). Так как

```
while(*p++=*q++) ;
```

выглядит более чем странно для программистов на большинстве языков (кроме C/C++), а на C и C++ подобного рода фрагменты встречаются отнюдь не редко, то стоит присмотреться к ним подробнее.

Сначала рассмотрим более традиционный способ копирования массива символов:

```
int length=strlen(q) ;
for(int i=0; i<=length; i++) p[i] = q[i] ;
```

Это расточительный способ работы. Длину строки с терминальным нулем легко можно определить непосредственно в процессе просмотра символов этой строки. А так получается, что мы читаем строку дважды: первый раз — чтобы определить длину строки, а второй раз для копирования ее символов. Исправляем этот недочет:

```
int i;
for(i=0; q[i]!=0; i++) p[i] = q[i] ;
p[i]=0;                    // терминальный нуль
```

Так как *p* и *q* — указатели, то можно избавиться от переменной *i*, применяемой для индексирования:

```
while(*q!=0)
{
  *p=*q;
  p++;                     // указывает на следующий символ
```

```
    ++;                      // указывает на следующий символ
}
*p = 0;                      // терминальный нуль
```

Поскольку постфиксные инкремент и декремент позволяют нам использовать значение и лишь потом изменить его, можно переписать цикл еще раз:

```
while (*q!=0)
{
    *p++=*q++;
}
*p=0;                        // терминальный нуль
```

Так как значение выражения *p++ = *q++ есть *q, то цикл можно записать еще короче:

```
while ((*p++=*q++) !=0) { }
```

Здесь равенство выражения *q нулю обнаруживается после того, как мы его копируем в *p и инкрементируем p, так что тем самым копируется и терминальный нуль (отдельное присваивание из предыдущего варианта кода становится ненужным). Окончательное сокращение кода достигается тем, что отбрасывается пустой блок, а также явное сравнение с нулем ввиду его избыточности. В итоге, мы приходим к первоначальному варианту кода:

```
while (*p++=*q++) ;
```

Можно ли сказать, что этот код менее читаем, чем предыдущие версии? Только не для опытных C и C++ программистов. А является ли он более эффективным? Не обязательно (если, конечно, не рассматривать версию с вызовом функции ***strlen***() для отдельного вычисления длины строки). Эффективность в сильной степени будет зависеть от архитектуры компьютера и особенностей конкретного компилятора.

Самый эффективный способ копирования строк с терминальным нулем на вашей машине должна обеспечивать соответствующая стандартная библиотечная функция:

```
char* strcpy(char*, const char*);      // из <string.h>
```

В более общих случаях можно применить *стандартный алгоритм* ***copy*** (§2.7.2, §18.6.1). Везде, где только можно, *пользуйтесь средствами стандартной библиотеки вместо возни с указателями и байтами*. Функции стандартной библиотеки могут быть встраиваемыми (§7.1.1) и даже реализовываться с помощью специализированных машинных инструкций. Поэтому подумайте дважды, прежде чем отдать предпочтение собственному рукотворному коду перед стандартными библиотечными средствами.

6.2.6. Свободная память

Время жизни именованных объектов определяется их областью видимости (§4.9.4). Однако часто нужны объекты, время жизни которых не зависит от области видимости, в которой они были созданы. Таковы, например, объекты, с которыми нужно работать после возврата из функции, где они были созданы. Этот тип объектов создается *операцией* ***new***, а их уничтожение выполняется *операцией* ***delete***. Па-

мять под эти объекты выделяется из так называемой «свободной памяти» (или из «кучи»; динамической памяти).

Посмотрим, как можно написать компилятор в стиле, который мы применили для написания программы-калькулятора (§6.1). Функции синтаксического анализа могли бы строить дерево выражений для его дальнейшего использования генератором кода:

```
struct Enode
{
  Token_value oper;
  Enode* left;
  Enode* right;
  // ...
};

Enode* expr (bool get)
{
  Enode* left=term (get) ;

  for ( ; ; )
    switch (curr_tok)
    {
      case PLUS:
      case MINUS:
      {
        Enode* n = new Enode;          // создать Enode в свободной памяти
        n->oper=curr_tok;
        n->left=left;
        n->right=term (true) ;
        left=n;
        break;
      }
      default:
        return left;                   // вернуть узел
    }
}
```

Генератор кода использует построенные узлы дерева, а затем удаляет их:

```
void generate (Enode* n)
{
  switch (n->oper)
  {
    case PLUS:
      // ...
      delete n;                        // удалить Enode из свободной памяти
  }
}
```

Объект, созданный операцией **new**, *существует до тех пор, пока он не будет явным образом уничтожен операцией* **delete**. Только после этого *память, занимаемая объектом, освобождается и может далее снова использоваться операциями* **new**. Обычно реализации C++ не гарантируют автоматического освобождения памяти

(«сборки мусора») с тем, чтобы их снова могли использовать операции ***new***. Поэтому я полагаю, что объекты, созданные операцией ***new***, удаляются вручную операцией ***delete***. Лишь в случае наличия в конкретной реализации C++ автоматического сборщика мусора можно обойтись без операций ***delete*** (§C.9.1).

Применять операцию ***delete*** можно *лишь к указателям, значения которых или установлены операцией* ***new***, *или равны нулю*. Если ***delete*** применяется к нулю, то не производится никаких действий.

Более специализированные версии операции ***new*** обсуждаются в §15.6.

6.2.6.1. Массивы

Массивы объектов также можно создавать операцией ***new***. Например:

```
char* save_string (const char* p)
{
  char* s=new char[strlen (p) +1] ;
  strcpy (s,p) ;                        // скопировать из p в s
  return s;
}

int main (int argc, char* argv[ ] )
{
  if(argc < 2) exit (1) ;
  char* p = save_string (argv[1] ) ;
  // ...
  delete[ ] p;
}
```

Операция ***delete*** применяется для удаления одиночных объектов; ***delete*** [ ] используется для удаления массивов.

Чтобы корректно освободить память, выделенную операцией ***new***, операции ***delete*** и ***delete*** [ ] должны знать размер удаляемого объекта. Отсюда следует, что размер памяти, динамически выделяемой под объект стандартной реализацией операции ***new***, несколько больше размера памяти, отводимого под статический объект. В типичном случае используется одно дополнительное слово для хранения размера объекта.

Так как объекты типа ***vector*** (§3.7.1, §16.3) ничем не хуже иных объектов, то их, естественно, можно динамически размещать в памяти операцией ***new*** и удалять оттуда (уничтожать) операцией ***delete***. Вот пример на эту тему:

```
void f(int n)
{
  vector<int>* p = new vector<int> (n) ;   // индивидуальный объект
  int* q = new int[n] ;                    // массив
  // ...
  delete p;
  delete[ ] q;
}
```

Применять операцию ***delete*** [ ] можно лишь к указателям на массивы, значения которых или установлены операцией ***new***, или равны нулю. Если ***delete*** [ ] применяется к нулю, то не производится никаких действий.

6.2.6.2. Исчерпание памяти

Реализации операций ***new***, ***delete***, ***new*** [] и ***delete*** [] используют следующие стандартные функции, прототипы которых представлены в заголовочном файле <***new***> ((§19.4.5):

```
void* operator new (size_t) ;          // выделяет память для индивидуального объекта
void* operator delete (void* p) ;      // при p!=0 освобождает выделенную операцией
                                       // new память

void* operator new[] (size_t) ;        // выделяет память под массив
void* operator delete[] (void* p) ;    // при p!=0 освобождает выделенную операцией
                                       // new[] память
```

Когда операции ***new*** требуется выделить память под некоторый объект, вызывается *функция* ***operator new***(), которая и выделяет нужное количество байт. Аналогично, когда операции ***new*** [] требуется выделить память под массив, вызывается *функция* ***operator new***[]().

Стандартные реализации функций ***operator new***() и ***operator new***[]() не инициализируют выделяемую ими память.

Что произойдет, когда операции ***new*** не удастся найти в свободной памяти блок затребованного размера? По умолчанию в этом случае генерируется исключение ***bad_alloc*** (другие варианты смотри в §19.4.5). Например:

```
void f()
{
  try
  {
    for(; ; ) new char[10000] ;
  }
  catch(bad_alloc)
  {
    cerr " "Memory exhausted!\n";
  }
}
```

Какова бы ни была память на вашей машине, здесь рано или поздно сработает обработчик исключения ***bad_alloc***.

У нас есть возможность явно указать, что должно происходить при исчерпании памяти операцией ***new***, так как в этот момент операция new вызывает функцию, зарегистрированную с помощью стандартной функции ***set_new_handler***() (ее прототип дан в заголовочном файле <***new***>), если, конечно, такая регистрация вообще имела место. Например:

```
void out_of_store()
{
  cerr<<"operator new failed: out of store\n";
  throw bad_alloc() ;
}

int main()
{
  set_new_handler(out_of_store) ; // делаем out_of_store обработчиком нехватки памяти
```

```
  for ( ; ; ) new char[10000] ;
  cout<<"done\n" ;
}
```

Эта программа выведет строку

operator new failed*: *out of store

В §14.4.5 приведена правдоподобная реализация функции ***operator new*()**, которая проверяет наличие зарегистрированного обработчика и генерирует исключение ***bad_alloc*** в случае отсутствия такового. Зарегистрированный обработчик может делать что-нибудь более умное, чем просто завершение программы. Например, обработчик может все же попытаться отыскать память, запрошенную операцией ***new***, реализовав некоторый вариант сборки мусора (операция ***delete*** станет в таком случае ненужной). Новичок, конечно, с такой задачей вряд ли справится, но можно приобрести готовый и оттестированный обработчик от сторонних производителей (§C.9.1).

Предоставляя новый обработчик, мы контролируем ситуацию с исчерпанием памяти при обращении к операции ***new***. Существуют два способа управления процессом выделения памяти. Можно либо предоставить нестандартные реализации функций ***operator new*()** и ***operator delete*()**, которые вызываются стандартной формой операции ***new***, либо положиться на информацию о блоке памяти, предоставленную пользователем (§10.4.11, §19.4.5).

6.2.7. Явное приведение типов

Иногда приходится иметь дело с *«сырой» памятью* (*raw memory*), то есть памятью, которая предназначена для хранения объектов неизвестного компилятору типа. К примеру, распределитель памяти возвращает значение типа ***void****, указывающее на выделенный блок памяти, или мы трактуем целое число как адрес устройства ввода/вывода:

```
void* malloc (size_t) ;

void f()
{
  int* p=static_cast<int*> (malloc (100) ) ;    // память выделена под целые

  IO_device* dI=
  reinterpret_cast<IO_device*> (0Xff00) ;       // устройство по адресу 0Xff00
  // ...
}
```

Компилятор не знает тип объекта, адресуемого указателем типа ***void****. Не может он и знать, корректен ли адрес ***0Xff00***. Следовательно, именно программист несет всю ответственность за корректность преобразований типов. *Явное преобразование типов* (*приведение типов — casting*) на практике важно и применимо. Но по установившейся традиции его применение избыточно и часто служит источником ошибок.

Операция ***static_cast*** осуществляет преобразования между родственными типами данных, например от указателя на некоторый класс к указателю на другой класс из той же иерархии наследования, от интегрального типа к перечислению, от типа

с плавающей запятой к интегральному типу. *Операция* ***reinterpret_cast*** осуществляет преобразования между несвязанными типами, например от целого к указателю или от одного указателя к иному произвольному указательному типу. Различие между этими операциями позволяет компилятору немного контролировать процесс приведения типов операцией ***static_cast*** и помогает программисту следить за потенциально опасными приведениями, выполняемыми в программе с помощью ***reinterpret_cast***. Преобразования типов с помощью ***static_cast*** *более-менее переносимы*, а преобразования через ***reinterpret_cast*** *— практически никогда*. Нельзя утверждать со 100%-ой уверенностью, но все же чаще всего ***reinterpret_cast*** дает новую трактовку типа, оставляя последовательности битов аргумента неизменными. Если конечный тип преобразования не уже, чем исходный (то есть содержит не меньшее число бит), то можно еще раз применить ***reinterpret_cast*** в обратном порядке и вернуться к исходному аргументу. В общем случае, результат операции ***reinterpret_cast*** гарантированно приемлем для использования лишь тогда, когда преобразуемое значение соответствует целевому типу.

Если вы решили выполнить явное приведение типа, то сначала задумайтесь, насколько оно действительно необходимо. В C++ необходимость в явных преобразованиях типов встречается существенно реже, чем в C (§1.6) или в ранних версиях C++ (§1.6.2, §B.2.3). Во многих программах можно полностью избавиться от явного преобразования типов. В других — тщательно локализовать их внутри немногочисленных процедур. В настоящей книге реальные случаи применения явных преобразований типов встречаются лишь в §6.2.7, §7.7, §13.5, §13.6, §17.6.2.3, §15.4, §25.4.1 и §E.3.1.

Имеются также операция преобразования ***dynamic_cast***, действие которой контролируется на этапе выполнения программы (§15.4.1), и операция преобразования ***const_cast*** (§15.4.2.1), аннулирующая действие *модификаторов* ***const*** и ***volatile***.

Язык C++ унаследовал от C конструкцию (***T***) ***e***, означающую любое преобразование, которое может быть выполнено комбинацией операций ***static_cast***, ***reinterpret_cast*** и ***const_cast***, для получения значения типа ***T*** из выражения ***e*** (§B.2.3). Такой стиль преобразований опаснее рассмотренных выше именованных операций приведения, ибо его труднее находить в больших программах и он не отражает точных намерений программиста, поскольку (***T***) ***e*** может выполнять и переносимое преобразование между родственными типами данных, и непереносимое преобразование несвязанных между собой типов и снятие действия модификатора ***const*** с указателя. Без точного знания типа ***T*** и типа выражения ***e*** ничего определенного сказать нельзя.

6.2.8. Конструкторы

Для *конструирования* (*создания*) значения типа ***T*** из значения e используется функциональная форма записи ***T***(***e***). Например:

```
void f(double d)
{
  int i=int(d) ;             // усекаем d (отбрасываем дробную часть)
  complex z=complex(d) ;     // создаем complex из d
  // ...
}
```

Иногда конструкцию ***T*(*e*)** называют *приведением типов в функциональном стиле* (*function-style cast*). К сожалению, для встроенного типа ***T*** запись ***T*(*e*)** эквивалентна записи **(*T*) *e*** (§6.2.7). Это означает, что для многих встроенных типов применение ***T*(*e*)** небезопасно. Например, значения арифметических типов могут оказаться *усеченными* (*truncated*). Даже явные преобразования от более длинных целых типов к более коротким (например, из ***long*** в ***char***) приводят к непереносимым, зависящим от реализации результатам. Я стараюсь применять конструкцию ***T*(*e*)** только в четко определенных случаях: для сужающих арифметических преобразований (§С.6), для приведения целых к перечислениям (§4.8) и для конструирования объектов пользовательских типов (§2.5.2, §10.2.3).

Преобразования указателей нельзя выполнять непосредственно в виде ***T*(*e*)**. К примеру, ***char** (2)* является синтаксической ошибкой. К сожалению, такую защиту против опасных приведений указателей можно обойти, используя синонимы указательных типов, вводимые с помощью ***typedef*** (§4.9.7).

Конструкция ***T*()** используется для создания значения по умолчанию для типа ***T***. Например:

```
void f(double d)
{
  int j=int () ;                  // умолчательное целое значение
  complex z=complex () ;          // умолчательное значение для типа complex
  // ...
}
```

Для встроенных типов значением по умолчанию является нуль, преобразованный в соответствующий тип (§4.9.5). Таким образом, ***int*()** есть просто другая форма записи нулевого целого значения. Для пользовательского типа ***T***, действие выражения ***T*()** определяется так называемым конструктором по умолчанию (если он имеется) (§10.4.2).

Роль конструирующих выражений для встроенных типов приобретает особую важность в случае определения шаблонов, когда программист не знает, будет ли шаблонный параметр соответствовать встроенному или пользовательскому типам данных (§16.3.4, §17.4.1.2).

6.3. Обзор операторов языка С++

Ниже приведена компактная сводка операторов языка С++.

Синтаксис операторов

оператор:
    объявление
    { *последовательность-операторов$_{opt}$* }
    try { *последовательность-операторов$_{opt}$* } *список-обработчиков*
    выражение$_{opt}$;
    if (*условие*) *оператор*
    if (*условие*) *оператор* ***else*** *оператор*

Синтаксис операторов
switch *(условие) оператор*
while *(условие) оператор*
do *оператор* ***while*** *(выражение) ;*
for *(инициализирующий-оператор условие$_{opt}$; выражение$_{opt}$) оператор*
case *константное выражение : оператор*
default*: оператор*
break;
continue;
return *выражение$_{opt}$;*
goto идентификатор;
идентификатор: оператор
последовательность-операторов:
оператор последовательность-операторов$_{opt}$,
условие:
выражение
спецификатор-типа объявитель = выражение
список-обработчиков:
catch *(объявление-исключения) { последовательность-операторов$_{opt}$ }*
список-обработчиков список-обработчиков$_{opt}$

Здесь $_{opt}$ означает «optional», то есть «необязательно».

Обратите внимание на то, что объявление является оператором, и что нет никаких операторов присваивания и операторов вызова функций; *присваивания и вызов функции — это выражения, использующие соответствующие операции.* Операторы для обработки исключений, так называемые ***try****-блоки* (*try-blocks*), будут рассмотрены в §8.3.1.

6.3.1. Объявления как операторы

Объявление является оператором. Если только переменная не объявлена с *модификатором* ***static***, то инициализатор выполняется всякий раз, как только поток управления проходит через соответствующее объявление (§10.4.8). Причина, по которой разрешается применять объявления всюду, где допускается оператор (и еще в некоторых местах; §6.3.2.1, §6.3.3.1), заключается в стремлении уменьшить количество ошибок, связанных с применением неинициализированных переменных, и для лучшей локализации кода. Редко когда возникают причины для объявления переменных ранее момента, когда становятся известными (доступными) их начальные значения. Например:

```
void f(vector<string>& v, int i, const char* p)
{
  if(p==0) return;
```

```
    if(i<0 || v.size()<=i) error("bad index");
    string s = v[i];
    if(s == p)
    {
      //...
    }
    //...
}
```

Возможность поместить объявление после некоторого количества исполняемого кода существенно для констант и вообще для стиля программирования, который можно назвать стилем однократных присваиваний, когда значения объектов после их инициализации не меняются. Для пользовательских типов с точки зрения эффективности кода также важно отложить определение объектов до момента появления приемлемого инициализатора. Например,

```
string s; /* ... */ s="The best is the enemy of the good.";
```

вполне может оказаться менее эффективным, чем

```
string s="Voltaire";
```

Наиболее общей причиной, по которой переменные объявляются без инициализации, является необходимость использования отдельных операторов для придания им конкретных значений. Например, когда значения переменных и массивов извлекаются из потока ввода.

6.3.2. Операторы выбора (условные операторы)

Значение выражения может проверяться в операторах ***if*** или ***switch***:

```
if(условие) оператор
if(условие) оператор else оператор
switch (условие) оператор
```

Операции сравнения

```
==   !=   <   <=   >   >=
```

возвращают логическое значение ***true***, если сравнение истинно, и логическое ***false*** в противном случае.

Для оператора ***if*** выполняется первый подчиненный оператор (statement), если значение проверяемого выражения (condition) не равно нулю, или второй подчиненный оператор (если он есть) в противном случае. Отсюда следует, что *любое арифметическое выражение или выражение с указателями может использоваться в качестве проверяемого выражения* (условия). Например, если ***x*** целое, то

```
if(x)    //...
```

означает

```
if(x!=0)  //...
```

Для указателя ***p***, следующий код

```
if(p)    //...
```

представляет собой прямую проверку «указывает ли *p* на действительный объект?», в то время как

```
if (p!=0) //...
```

делает то же самое, но косвенно, сравнивая *p* с нулевым значением, которое, как известно, не может быть корректным адресом объекта в памяти. Заметим, что «нулевой» указатель не на всех машинах представляется битовой последовательностью из одних нулей (§5.1.1). Все проверенные мной компиляторы для обеих форм сравнения генерировали одинаковый код.

Логические операции

```
&& || !
```

чаще всего используются именно в проверяемых выражениях (условиях). *Операции* && и || *вообще не вычисляют свой второй* (правый) *аргумент, если в этом отсутствует необходимость.* Например,

```
if( p && 1<p->count ) //...
```

сначала проверяет *p* на равенство нулю. Выражение *1<p->count* вычисляется (проверяется) только в случае, когда *p* не равен нулю.

Некоторые условные операторы (операторы *if*) легко могут быть заменены на *условные выражения* (*conditional-expressions*). Например,

```
if(a<=b)
  max=b;
else
  max=a;
```

лучше выразить следующим образом:

```
max= (a<=b) ? b:a;
```

Здесь круглые скобки вокруг проверяемого условия не обязательны, но мне кажется, что с ними код читается лучше.

Оператор **switch** можно заменить на эквивалентный набор операторов *if*. К примеру,

```
switch (val)
{
  case 1:
    f();
    break;
  case 2:
    g();
    break;
  default:
    h();
    break;
}
```

можно представить и как

```
if(val==1)
  f();
else if(val==2)
```

```
  g();
else
  h();
```

Смысл тот же самый, но первая форма (оператор ***switch***) предпочтительнее, ибо явным образом отражает центральную идею о сравнении значения выражения с набором констант. Это делает оператор ***switch*** лучше читаемым в сложных случаях, и даже генерируемый машинный код может оказаться более эффективным. Обратите внимание на то, что каждая case-ветвь оператора ***switch*** должна специальным образом завершаться, если только вы не хотите, чтобы продолжали работать и последующие ветви. Например,

```
switch (val)
{
  case 1:
    cout<< "case 1\n";
  case 2:
    cout<< "case 2\n";
  default:
    cout<< "default: case not found\n";
}
```

при ***val==1*** выдаст (к изумлению непосвященных) следующий результат:

```
case 1
case 2
default: case not found
```

Случаи, когда такой результат соответствует намерению программиста, лучше специально комментировать, чтобы легче было по отсутствию комментариев находить участки кода, в которых подобного рода эффекты не планировались, но ошибочным образом все же состоялись. Чаще всего для завершения ***case***-ветвей оператора ***switch*** используется оператор ***break***, но также применяется и оператор ***return*** (§6.1.1).

6.3.2.1. Объявления в условиях

Во избежание случайного неправильного использования переменных их лучше объявлять в максимально узких областях видимости. В частности, лучше откладывать определение локальной переменной до появления подходящего для нее начального значения, что автоматически предотвращает непреднамеренное использование этой переменной до инициализации.

Элегантное применение этих идей заключается в *объявлении переменной внутри условий*. Рассмотрим пример:

```
if(double d =prim(true))
{
  left /= d;
  break;
}
```

Здесь ***d*** объявляется и инициализируется, после чего проверяется в качестве условия. *Область видимости переменной **d** начинается в точке ее объявления и распростра-*

няется на все блоки, контролируемые условием. Например, будь в данном примере еще и ***else***-ветвь оператора ***if***, область видимости ***d*** распространялась бы на обе ветви.

Традиционной альтернативой является объявление ***d*** до условия. Однако это открывает лазейки для использования переменной ***d*** в неинициализированном состоянии, а также вне предназначенной для нее области действия:

```
double d;
// ...
d2 = d;        // oops!
// ...
if (d = prim (true) )
{
  left /= d;
  break;
}
// ...
d = 2.0;       // два разных использования d
```

В дополнение к рассмотренным преимуществам объявление переменной в условиях приводит еще и к более компактному исходному коду.

Объявление и инициализация в условиях может распространяться лишь на единственную переменную или ***константу***.

6.3.3. Операторы цикла

Циклическое выполнение кода реализуется операторами ***for***, ***while*** и ***do***:

```
while (условие) оператор
do оператор while (выражение) ;
for (инициализирующий_оператор условие_opt ; выражение_opt) оператор
```

Каждый из перечисленных управляющих операторов обеспечивает повторное выполнение оператора ***statement*** (называемого управляемым оператором или телом цикла) до тех пор, пока проверяемое выражение (condition или expression) не примет значения ***false*** или пока программист не прервет цикл каким-либо иным способом.

Оператор ***for*** предназначен для стандартной организации циклов. В этом операторе переменная цикла, условие окончания и выражение для изменения переменной цикла компактно записываются в единственной строке в самом начале цикла. Это в сильнейшей степени увеличивает читаемость кода и, тем самым, уменьшает вероятность ошибок. В случае отсутствия необходимости в инициализации цикла инициализирующую часть (*for-init-statement*) можно опустить. Если опущено условие окончания, то цикл будет выполняться вечно, если только программист не предусмотрит его окончание за счет применения операторов ***break***, ***return***, ***goto***, ***throw*** или менее очевидными способами, такими как вызов стандартной функции ***exit***() (§9.4.1.1). Если опущено выражение для изменения переменной цикла, то эту задачу нужно выполнить в теле цикла. Если *проектируемый цикл* не вписывается в каноническую схему «ввел переменную цикла, проверил условие окончания, изменил значение переменной цикла», то лучше применить *оператор* ***while***. *Цикл* ***for*** можно применить для записи бесконечного цикла (с отсутствием явно заданных условий окончания):

```
for ( ; ; )            // "вечно"
{
  // ...
}
```

*Цикл **while*** заставляет управляемый оператор исполняться до тех пор, пока условие цикла не станет *ложным*. Я предпочитаю использовать *цикл **while*** вместо *цикла **for*** в случаях, когда нет очевидной переменной цикла или когда ее модификацию лучше выполнять где-нибудь внутри тела цикла. Очевидным примером цикла без явной переменной цикла является цикл ввода:

```
While (cin>>ch)   // ...
```

Мой опыт показывает, что *цикл **do*** скорее является источником ошибок и недоразумений, чем острой необходимостью. Единственным поводом к его применению является тот факт, что тело этого цикла всегда исполняется хотя бы один раз до самой первой проверки условия цикла. Но для надежного выполнения первой итерации все равно должно проверяться некоторое условие. Я обнаружил, что чаще, чем это можно предположить, условие корректности первого прохода цикла либо не обеспечивалось с самого начала, либо переставало соблюдаться после модификации текста программы. Также, на мой взгляд, предпочтительнее формулировать условие окончания цикла в его начале, где оно бросается в глаза и его можно проанализировать. В общем, я предпочитаю *циклов **do*** не применять.

6.3.3.1. Объявления в операторах цикла for

*В инициализирующей части оператора **for** можно объявлять переменные.* Область видимости объявленной таким образом переменной (переменных) распространяется *до конца цикла*. Например:

```
void f(int v[ ] , int max)
{
 for (int i=0; i<max; i++) v[i]=i*i;
}
```

Если конечное значение переменной цикла нужно использовать после его окончания, то переменную цикла следует объявить вне оператора *цикла **for*** (§6.3.4).

6.3.4. Оператор goto

В языке С++ пресловутый оператор ***goto*** сохранен:

goto *идентификатор;*
идентификатор: оператор

Оператор ***goto*** полезен в ряде случаев и в обычном высокоуровневом программировании, но он особо полезен, когда программу создает не человек, а другая программа: например, при автоматической генерации парсера на базе формальной грамматики. Также ***goto*** полезен для программ реального времени, когда нужно с высокой эффективность реализовать выход из внутреннего цикла.

Областью действия *меток* (*labels*) является *тело функции*, в *котором они находятся*. Отсюда следует, что возможны переходы (по ***goto***) внутрь блоков и из блоков на-

ружу. Только при этом нельзя перепрыгивать через объявления с инициализациями и непосредственно внутрь обработчиков прерываний (§8.3.1).

Важным примером использования ***goto*** в обычном коде является организация с его помощью выхода из вложенных циклов или операторов ***switch*** (оператор ***break*** обеспечивает выход лишь из одного уровня вложенности). Например:

```
void f()
{
  int i;
  int j;
  for (i = 0; i<n; i++)
    for (j = 0; j<m; j++) if (nm[i] [j] == a) goto found;
      // not found
      // ...
found:
      // nm[i][j] == a
}
```

Имеется также *оператор **continue***, который *передает управление в конец тела цикла* (§6.1.5).

6.4. Комментарии и отступы

Разумное применение комментариев и согласованная система отступов могут сделать процесс чтения программы более приятным и способствовать более быстрому ее изучению и пониманию. Имеется несколько общеупотребительных стилей для отступов строк кода. Я не вижу никаких объективных причин для предпочтения какого-либо из этих стилей (хотя у меня, как и у любого программиста, есть свой привычный стиль). То же самое можно сказать и о стилях комментирования.

Вообще говоря, комментариями можно даже ухудшить читаемость программы. Компилятор существа комментариев не понимает и не может гарантировать, что комментарий:

1. Содержателен.
2. Имеет отношение к программе.
3. Не устарел.

Многие программы содержат комментарии, понять которые невозможно, которые противоречивы и даже просто неверны. Плохие комментарии хуже их отсутствия.

Если что-то можно отразить явным образом средствами языка, то так и нужно делать, а не ограничиваться простым упоминанием в комментариях. Вот примеры на эту тему:

```
// переменную v нужно проинициализировать
// переменную v должна использовать лишь функция f()
// вызовите функцию init() до вызова любой другой функции из этого файла
// вызовите функцию cleanup() в конце вашей программы
// не пользуйтесь функцией weird()
// функция f() принимает два аргумента
```

При правильном кодировании на C++ подобного рода комментарии просто не нужны. Вместо них лучше полагаться на правила компоновки (§9.2) и области видимости, на инициализацию и деинициализацию объектов классов (§10.4.1).

Если что-то ясно выражено средствами языка, не надо это повторять в комментариях. Например:

```
a=b+c;       // a принимает значение, равное b+c
count++;     // инкрементируем переменную counter
```

Такие комментарии не только не нужны, но они еще и вредны. Они увеличивают совокупный объем кода, подлежащий чтению, они зачастую затуманивают структуру программы и могут быть просто неверными. Заметим, однако, что такие комментарии интенсивно используются в учебниках, и в данной книге в том числе. Этим, помимо прочего, программы из учебников отличаются от реальных профессиональных программ.

Я предпочитаю применять комментарии в следующих случаях:

1. В начале каждого исходного файла — комментарии, поясняющие, что общего у всех представленных в файле объявлений, ссылки на литературу и другие источники, соображения по поводу дальнейшего сопровождения и т.д.
2. Комментарии к классам, шаблонам, пространствам имен.
3. Комментарии к каждой нетривиальной функции, поясняющие ее назначение, алгоритмы (если они не очевидны), и, возможно, некоторые предположения о контексте вызова.
4. Комментарии к переменным и константам из глобального или именованного пространств имен.
5. Комментарии к неочевидным или непереносимым участкам кода.
6. Редко в иных случаях.

Например:

```
//   tbl.c: Реализация таблицы символов.
/*
     Исключение методом Гаусса.
     См. Ralston: "A first course ..." pg 411.
*/
//   swap() предполагает раскладку стека как в SGI R6000.
/************************************

     Copyright (c) 1997 AT&T, Inc.
     All rights reserved

************************************/
```

Хорошо продуманные и хорошо написанные комментарии являются неотъемлемой частью хороших программ. Написание таких комментариев столь же сложно, как и написание собственно кода программы. Стоит развивать и культивировать искусство написания хороших комментариев.

Отметим, что если в функции применяются исключительно комментарии вида `//`, то целые фрагменты ее кода можно закомментировать (и быстро раскомментировать) с помощью `/* */`.

6.5. Советы

1. Предпочитайте всем библиотекам и «самодельному коду» стандартную библиотеку; §6.1.8.
2. Избегайте слишком сложных выражений; §6.2.3.
3. Используйте круглые скобки в случае сомнений по поводу приоритетов операций; §6.2.3.
4. Избегайте явных приведений типов; §6.2.7.
5. Если явное приведение типов необходимо, используйте специфические (именованные) операции приведения вместо операций в С-стиле; §6.2.7.
6. Используйте конструкции *T(e)* только тогда, когда правила конструирования четко определены; §6.2.8.
7. Избегайте выражений с неопределенным порядком вычисления; §6.2.2.
8. Избегайте применения оператора ***goto***; §6.3.4.
9. Избегайте применения *циклов* ***do***; §6.3.3.
10. Не объявляйте переменные до того, как станут известны инициализирующие их значения; §6.3.1, §6.3.2.1, §6.3.3.1.
11. Пишите ясные и краткие комментарии; §6.4.
12. Придерживайтесь согласованного стиля отступов; §6.4.
13. Для переопределения глобальной функции ***operator new***() применяйте члены классов (§15.6); §6.2.6.2.
14. При чтении ввода всегда помните о возможных «сюрпризах»; §6.1.3.

6.6. Упражнения

1. (*1) Перепишите следующий цикл ***for*** в виде эквивалентного ***while*** цикла:

```
for(i=0; i<max_length; i++)
  if(input_line[i]=='?') quest_count++;
```

   Перепишите так, чтобы проверяемой величиной был указатель и условие цикла имело вид `*p=='?'`.

2. (*1) Расставьте скобки в следующих выражениях:

```
a = b + c* d<<2 & 8
a& 077! = 3
a == b||a == c && c< 5
c = x! = 0
0<= i<7
f(1,2)+3
a = - 1++ b -- 5
a = b == c++
a = b = c = 0
a[4][2] *= * b ? c : * d* 2
a-b,c=d
```

3. (*2) Введите последовательность возможно разделенных пробельными символами пар (имя, значение). Имя — единственное слово, ограниченное пробельными символами. Значение формируется целым числом или числом с плавающей запятой. Вычислите и выведите сумму и среднее как для каждого отдельного имени, так и для всех имен (см. §6.1.8).
4. (*1) Напишите таблицу результатов всех побитовых логических операций (§6.2.4) для всех возможных комбинаций операндов *0* и *1*.
5. (*1.5) Приведите 5 конструкций языка C++, смысл которых не определен (§C.2); (*1.5) Приведите 5 конструкций языка C++, смысл которых зависит от реализации (§C.2).
6. (*1) Приведите 10 примеров непереносимого кода на C++.
7. (*2) Напишите 5 выражений, порядок вычисления которых не определен. Выполните код, чтобы посмотреть, что при этом реально делается в разных реализациях.
8. (*1.5) Что происходит при делении на нуль в вашей системе? Что происходит при переполнении (или потере точности)?
9. (*1) Расставьте скобки в следующих выражениях:

```
*p++
*--p
++a--
(int*)p->m
*p.m
*a[i]
```

10. (*2) Напишите следующие функции: *strlen*(), которая возвращает длину строк в C-стиле; *strcpy*(), которая копирует содержимое одной C-строки в другую; *strcmp*(), которая сравнивает содержимое двух C-строк. Решите, какого типа должны быть аргументы и возвращаемые значения. Затем сравните ваши варианты со стандартными библиотечными версиями, объявленными в <*cstring*> (<*string.h*>) и рассмотренными в §20.4.1.
11. (*1) Проверьте, как компилятор реагирует на ошибки в следующем фрагменте:

```
void f(int a, int b)
{
  if(a=3)          //...
  if(a&077==0)     //...
  a := b+1;
}
```

Придумайте еще несколько ошибок и проверьте реакцию компилятора на них.

12. (*2) Модифицируйте программу из §6.6[3], чтобы она вычисляла и медиану.
13. (*2) Напишите функцию *cat*(), которая принимает в качестве аргументов две C-строки и возвращает конкатенированную C-строку. Используйте операцию *new* для выделения памяти под результат.
14. (*2) Напишите функцию *rev*() для реверсирования содержимого C-строки.

15. (*1.5) Что делает следующий пример?

```
void send(int* to, int* from, int count)
// Полезные комментарии умышленно удалены.
{
 int n = (count+7)/8;
 switch(count%8)
  {
   case 0:  do{ *to++=*from++;
   case 7:      *to++=*from++;
   case 6:      *to++=*from++;
   case 5:      *to++=*from++;
   case 4:      *to++=*from++;
   case 3:      *to++=*from++;
   case 2:      *to++=*from++;
   case 1:      *to++=*from++;
              } while(--n>0);
  }
}
```

Зачем кому-то может потребоваться подобный код?

16. (*2) Напишите функцию ***atoi***(***const char****), которая принимает С-строку, содержащую цифры и возвращает соответствующее целое. Например, для ***atoi***("***123***") должно получиться целое число *123*. Модифицируйте функцию так, чтобы помимо десятичных чисел обрабатывались также восьмеричные и шестнадцатеричные.
17. (*2) Напишите функцию ***itoa***(***int i***, ***char b***[]), которая формирует строковое представление ***i*** в ***b*** и возвращает ***b***.
18. (*2) Наберите текст программы-калькулятора и заставьте его работать. Не экономьте время, применяя заранее приготовленный кем-то текст. Вы многому научитесь, отыскивая и исправляя «глупые мелкие ошибки».
19. (*2) Модифицируйте программу-калькулятор, чтобы она сообщала номера строк, в которых произошла ошибка.
20. (*3) Разрешите пользователю определять функции для программы-калькулятора. Подсказка: определите функцию в виде последовательности операций в порядке, в котором их ввел пользователь. Последовательность можно хранить либо в виде строки, либо как список лексем. При вызове функции читайте и выполняйте эти операции. Если допускаются аргументы у пользовательских функций, придумайте соответствующую форму записи.
21. (*1.5) Модифицируйте программу-калькулятор так, чтобы вместо статических переменных ***number_value*** и ***string_value*** использовалась структура ***symbol***.
22. (*2.5) Напишите код, очищающий программу на C++ от комментариев. Пусть она читает из ***cin***, удаляет комментарии вида // и /* */ и записывает результат в ***cout***. Не заботьтесь о внешнем виде результирующей программы (это была бы более сложная задача). Не заботьтесь о корректности программ. Не забудьте про символы //, /* и */ в комментариях, строках и символьных константах.
23. (*2) Выполните просмотр ряда программ, чтобы составить представление о применяемых стилях отступов, именования и комментирования.

7

Функции

Итерация — от человека,
рекурсия — от Бога.
— Л. Питер Дойч

Объявления и определения функций — передача аргументов — возвращаемые значения — перегрузка функций — разрешение неоднозначностей — аргументы по умолчанию — неуказанное число аргументов — указатели на функции — макросы — советы — упражнения.

7.1. Объявления функций

Обычно, сделать что-либо в C++ означает вызвать функцию. Определить функцию — значит задать способ выполнения работы. Функцию нельзя вызвать, если она не была предварительно объявлена.

Объявление функции (*function declaration*) предоставляет ее *имя*, *тип возвращаемого значения* (если оно есть), а также *число и типы аргументов*, которые требуется передавать функции при ее вызове. Например:

```
Elem* next_elem();
char* strcpy(char* to, const char* from);
void exit(int);
```

Семантика передачи аргументов идентична семантике инициализации. Производится проверка типов аргументов и при необходимости выполняется неявное приведение типов. Например:

```
double sqrt(double);

double sr2=sqrt(2);          // вызов sqrt() с аргументом double(2)
double sr3=sqrt("three");    // error: sqrt() требует аргумент типа double
```

Значение таких проверок и преобразований типов не следует недооценивать.

Объявления функций могут содержать имена аргументов. Это полезно программисту, читающему код, но компилятор их просто игнорирует. Как упоминалось

в §4.7 тип ***void*** для возвращаемого значения функции фактически означает его отсутствие.

7.1.1. Определения функций

Любая вызываемая в программе функция должна быть где-то определена, причем лишь один раз. *Определение функции* (*function definition*) есть ее объявление плюс тело функции. Например:

```
extern void swap (int* , int* ) ;  // объявление

void swap (int* p, int* q)         // определение
{
  int t=*p;
  *p = *q;
  *q = t;
}
```

Все типы в определении функции и ее объявлениях *обязаны совпадать*. Однако имена аргументов совпадать не обязаны, так как они не являются частью типа.

Нередки случаи, когда в определении функции часть аргументов не используется:

```
void search (table* t, const char* key, const char* )
{
  // третий аргумент не используется
}
```

Как показано в данном примере, *неиспользуемому аргументу можно не давать имени* вообще. Обычно, неименованные аргументы в определениях функций появляются либо вследствие произведенных упрощений, либо в качестве резерва на будущее. В обоих случаях само наличие реально неиспользуемого аргумента в определении функции позволяет не трогать при этом клиентский код, вызывающий функцию.

Можно определить функцию с модификатором ***inline***. Например:

```
inline int fac (int n)
{
  return (n<2) ? 1 : n*fac(n-1) ;
}
```

Такие функции называют *встраиваемыми* (*inline functions*), так как модификатор ***inline*** указывает компилятору, что он должен пытаться осуществлять прямое встраивание кода функции ***fac***() во все места, где эта функция вызывается, а не реализовывать этот код в единственном экземпляре, доступ к которому выполняется через стандартный механизм вызова. Умный компилятор может сгенерировать константу ***720*** на месте вызова ***fac***(***6***). Из-за того, что могут быть объявлены встраиваемыми рекурсивные или взаимно рекурсивные функции, гарантировать реальное встраивание кода функции в каждое место ее вызова невозможно. В зависимости от степени интеллектуальности конкретного компилятора мы может реально получить для нашего примера и ***720***, и ***6*fac***(***5***), и просто обычный вызов ***fac***(***6***).

Чтобы обеспечить возможность встраивания в случае рядового (а не сверхинтеллектуального) компилятора и компоновщика, нужно размещать определение (а не

только объявление) функции с модификатором ***inline*** в той же самой области видимости (§9.2). Модификатор ***inline*** не влияет на семантику функции. В частности, встраиваемые функции по-прежнему имеют уникальный адрес, так же как и их статические переменные (§7.1.2).

7.1.2. Статические переменные

Локальные переменные инициализируются всякий раз, как только поток исполнения достигает точки их определения. Это происходит при каждом вызове функции и каждый такой вызов располагает собственной копией локальной переменной. Если же локальная переменная объявляется с модификатором ***static***, то единственный, статически размещаемый в памяти объект (§С.9) будет использоваться для представления этой переменной во всех вызовах функции. Такая переменная будет инициализироваться лишь однажды, когда поток управления первый раз достигнет точки ее определения. Например:

```
void f(int a)
{
 while(a--)
 {
   static int n = 0;    // инициализируется один раз
   int x=0;             // иниц-ся 'a' раз при каждом вызове f()

   cout << "n == "<< n++ << ", x == " << x++ << '\n';
 }
}

int main()
{
 f(3);
}
```

В результате будет получен следующий вывод:

```
n == 0, x == 0
n == 1, x == 0
n == 2, x == 0
```

Статические переменные наделяют функции «долговременной памятью» без необходимости применения глобальных переменных, доступ к которым из других функций способен привести к их порче (§10.2.4).

7.2. Передача аргументов

При вызове функции выделяется память под ее формальные аргументы, и каждый формальный аргумент инициализируется значением соответствующего фактического аргумента. Семантика передачи аргументов идентична семантике инициализации. В частности, типы фактических параметров проверяются относительно типов формальных параметров, и в случае необходимости выполняются либо преобразования стандартных типов, либо преобразования, определенные для пользовательских типов. Существуют специальные правила для передачи массивов

(§7.2.1), средства для передачи аргументов без проверки типов (§7.6) и средства, предназначенные для задания аргументов по умолчанию (§7.5). Рассмотрим следующую функцию:

```
void f(int val, int& ref)
{
  val++;
  ref++;
}
```

Когда осуществляется вызов *f*(), выражение ***val***++ инкрементирует локальную копию первого фактического аргумента, в то время как выражение ***ref***++ инкрементирует непосредственно второй фактический аргумент. Например,

```
void g()
{
  int i=1;
  int j=1;
  f(i,j);
}
```

увеличит ***j***, но не ***i***. Первый аргумент, ***i***, передается *по значению* (*by value*), а второй аргумент, ***j***, передается *по ссылке* (*by reference*). Как упоминалось в §5.5, функции, которые модифицируют передаваемые им по ссылке аргументы, делают программу менее ясной и их следует, чаще всего, избегать (однако, см. §21.3.2). Правда, передача параметров большого размера по ссылке намного эффективнее их передачи по значению. В таком случае следует объявлять аргументы с модификатором ***const***, дабы явным образом подчеркнуть, что передача аргументов по ссылке осуществляется исключительно ради эффективности, а не для того, чтобы функция могла их модифицировать:

```
void f(const Large& arg)
{
  // здесь arg невозможно изменить без явного приведения типа
}
```

Отсутствие модификатора ***const*** в объявлении передаваемого по ссылке аргумента должно восприниматься как явно выраженное намерение модифицировать этот аргумент в теле функции:

```
void g(Large& arg);          // предполагается, что g() может изменить arg
```

Аналогично, объявляя аргумент функции как указатель на константу, мы декларируем, что значение объекта, на который ссылается этот указатель, не будет модифицироваться в этой функции. Например:

```
int strlen(const char*);                   // количество символов в С-строке
char* strcpy(char* to, const char* from);  // копирование С-строк
int strcmp(const char*, const char*);      // сравнение С-строк
```

Практическая роль аргументов с модификатором ***const*** возрастает с ростом размера программы.

Следует отметить, что семантика передачи аргументов отличается от семантики присваивания. Это имеет значение для передачи аргументов с модификатором

const, для передачи аргументов по ссылке и для аргументов некоторых пользовательских типов (§10.4.4.1).

Литералы, константы и аргументы, требующие преобразования типов, могут передаваться в функции с *const&* аргументами, и не могут передаваться в функции с не-*const&* аргументами. Разрешение преобразования для аргументов, объявленных как *const T&*, гарантирует передачу любых значений типа *T* через временные объекты (в случае необходимости). Например:

```
float fsqrt (const float&) ;  // sqrt в Fortran-стиле с аргументом-ссылкой

void g (double d)
{
  float r=fsqrt (2.0) ;       // перелача ссылки на временную перем-ю со значением 2.0f
  r=fsqrt (r) ;               // передача ссылки на r
  r=fsqrt (d) ;               // перелача ссылки на временную перем-ю со значением float(d)
}
```

Запрещение преобразований для не-*const&* аргументов (§5.5) устраняет глупые ошибки, порождаемые созданием временных объектов. Например:

```
void update (float& i) ;

void g (double d, float r)
{
  update (2.0f) ;             // error: константный аргумент
  update (r) ;                // передача ссылки на r
  update (d) ;                // error: требуется приведение типа
}
```

Будь эти вызовы разрешены, *update*() молча изменила бы временные переменные, которые тут же были бы удалены. Для программиста это было бы, скорее всего, неприятным сюрпризом.

7.2.1. Массивы в качестве аргументов

Если аргумент функции объявлен как массив, то при вызове ей передается *указатель на первый элемент массива.* Например:

```
int strlen (const char*) ;

void f()
{
  char v[] = "an array";
  int i = strlen (v) ;
  int j = strlen ("Nicholas") ;
}
```

Таким образом, аргумент типа *T*[] при передаче преобразуется к типу *T**. Отсюда следует, что присваивание значения элементу массива-аргумента означает изменение элемента самого массива (а не копии элемента массива). Другими словами, массивы отличаются от иных типов тем, что их нельзя передать в функцию по значению.

Так как размер массива неизвестен в вызываемой функции, то это могло бы стать неудобством, но есть способы обойти проблему. Для С-строк имеется терми-

нальный нуль, и длина строки, тем самым, легко вычисляется. Для других массивов размер можно передавать дополнительным аргументом. Например:

```
void compute1 (int* vec_ptr, int vec_size) ;  // один способ
struct Vec
{
  int* ptr;
  int size;
};

void compute2 (const Vec& v) ;                // другой способ
```

В качестве альтернативы вместо массивов можно использовать тип **vector** (§3.7.1, §16.3) из стандартной библиотеки.

С многомерными массивами проблем больше (§C.7), но часто вместо них можно применить массив указателей, который не требует специального обращения. Например:

```
char* day[]={"mon", "tue", "wed", "thu", "fri", "sat", "sun"};
```

Но опять-таки, тип **vector** и другие типы стандартной библиотеки являются прекрасной альтернативой низкоуровневым встроенным типам — массивам и указателям.

7.3. Возвращаемое значение

Функция *должна* возвращать значение, если только ее возврат не объявлен как **void** (функция **main**() является исключением — см. §3.2). И наоборот — функция *не может* возвращать значение, если она объявлена с ключевым словом **void**. Например:

```
int f1 () { }                  // error: не возвращается значение
void f2 () { }                 // ok
int f3 () {return 1; }         // ok
void f4 () {return 1; }        // error: возврат в void функции
int f5 () {return; }           // error: отсутствует возвращаемое значение
void f6 () {return; }          // ok
```

Возвращаемое значение задается в *операторе* **return**. Например:

```
int fac (int n) {return (n>1) ? n*fac (n-1) :1; }
```

Функцию, которая *вызывает саму себя*, называют *рекурсивной* (*recursive*).
Функция может содержать более одного оператора return:

```
int fac2 (int n)
{
  if(n>1) return n*fac2 (n-1) ;
  return 1;
}
```

Как и семантика передачи аргументов, *семантика возврата значения из функции идентична семантике инициализации.* Оператор return инициализирует *неименованную переменную*, имеющую тип возврата функции. Тип выражения, который содер-

жится в операторе return, сопоставляется с объявленным типом возврата функции, и при необходимости выполняются стандартные преобразования типов, или преобразования, определенные пользователем. Например:

```
double f()
{
  return 1;    // 1 неявно преобразуется в double(1)
}
```

Каждый раз при вызове функции выделяется память под ее аргументы и локальные (автоматические) переменные. После возврата из функции эта память считается свободной и может использоваться повторно. Поэтому указатель на локальную переменную возвращать не следует: значение, на которое он указывает, может измениться неожиданно:

```
int* fp() { int local=1; /* ... */ return &local; }   // плохо
```

Эта ошибка встречается реже эквивалентной ошибки с возвратом ссылок:

```
int& fr() {int local=1; /* ... */ return local; }    // плохо
```

К счастью, компилятор обычно предупреждает, что осуществляется возврат ссылки на локальную переменную.

Функция, объявленная с ключевым словом ***void***, не может возвращать значений. Поэтому в теле ***void***-функций можно использовать операторы ***return*** с выражениями вызова других ***void***-функций. Например:

```
void g(int* p) ;
void h(int* p) { /* ... */ return g(p) ; }       // ok: возвращает "никакое значение"
```

Подобного вида операторы return широко применяются при написании шаблонных функций, в которых тип возвращаемого значения является параметром шаблона (§18.4.4.2).

7.4. Перегрузка имен функций

Чаще всего, разным функциям дают разные имена. Однако когда концептуально одинаковые функции выполняют одинаковую работу над объектами разных типов, то удобно дать им одинаковые имена. Это называют *перегрузкой имен функций* (*overloading*). Такая техника всегда применялась для встроенных операций, например, одно и то же имя (обозначение) операции сложения, +, используется для сложения целых чисел, чисел с плавающей запятой и указательных типов. Эта идея легко распространяется на функции, определяемые пользователем. Например:

```
void print(int) ;                  // печать целого
void print(const char*) ;          // печать С-строки
```

С точки зрения компилятора у этих функций есть только одна общая черта — имя. Конечно, предполагается, что такие функции должны быть похожи по смыслу, но сам язык не накладывает здесь никаких ограничений, так что в этом вопросе он программисту и не помогает, и не мешает. Таким образом, перегрузка имен функций является просто удобным и наглядным способом именования, особенно для

функций с *расхожими именами*, такими как ***sqrt***, ***print*** или ***open***. Когда выбор имени важен с точки зрения семантики, возможность перегрузки имен функций становится ценной вдвойне. Это имеет место, например, для операций +, *, << и т.д., для конструкторов (§11.7) и в обобщенном программировании (§2.7.2, глава 18). Когда вызывается функция ***f()***, компилятор должен решить, какая именно из функций с этим именем должна быть выбрана. Для выбора компилятор сравнивает типы фактических аргументов вызова с типом формальных параметров всех функций с именем ***f***. Идея состоит в том, чтобы вызвать ту функцию, чьи аргументы подходят больше всего, или выдать ошибку компиляции в случае невозможности такого выбора. Например:

```
void print(double);
void print(long);

void f()
{
  print(1L);      // print(long)
  print(1.0);     // print(double)
  print(1);       // error: неоднозначность: print(long(1)) или print(double(1))?
}
```

Поиск наиболее подходящей для вызова функции из имеющегося набора перегруженных функций выполняется выбором единственного наилучшего соответствия между типами выражений для аргументов вызова и типами формальных параметров этих функций. Чтобы формализовать понятие наилучшего соответствия, следующие критерии применяются в указанном порядке:

1. Точное совпадение типов, при котором или вообще не нужно выполнять преобразований типов, или только самые тривиальные (имя массива к указателю на первый элемент, имя функции к указателю на функцию, тип ***T*** к типу ***const T***).
2. Совпадение после «продвижения типов вверх»: для интегральных типов это продвижения ***bool*** в ***int***, ***char*** в ***int***, ***short*** в ***int*** и их ***unsigned*** аналоги (§C.6.1), и продвижение ***float*** в ***double***.
3. Совпадение после стандартных преобразований типов: например, ***int*** в ***double***, ***double*** в ***int***, ***double*** в ***long double***, указатели на производные типы в указатели на базовые типы (§12.2), ***T**** в ***void**** (§5.6), ***int*** в ***unsigned int*** (§C.6).
4. Совпадение после преобразований типов, определяемых пользователем (§11.4).
5. Совпадение типов из-за *многоточий* (... — *ellipsis*) в объявлении функции (§7.6).

Если выявляются два совпадения одного и того же наивысшего уровня, то вызов функции считается *неоднозначным* (*ambiguous*) и отвергается компилятором. Сформулированные *критерии разрешения перегрузки* (*resolution rules*) основаны на соответствующих правилах языков C и C++ для встроенных числовых типов данных (§C.6). Например:

```
void print(int);
void print(const char*);
void print(double);
void print(long);
void print(char);
```

```
void h (char c, int i, short s, float f)
{
  print (c) ;       // точное соответствие:          print(char)
  print (i) ;       // точное соответствие:          print(int)
  print (s) ;       // интегральное продвижение:     print(int)
  print (f) ;       // продвижение от float к double: print(double)
  print ('a') ;     // точное соответствие:          print(char)
  print (49) ;      // точное соответствие:          print(int)
  print (0) ;       // точное соответствие:          print(int)
  print ("a") ;     // точное соответствие:          print(const char*)
}
```

Для вызова ***print*(*0*)** выбирается вариант ***print*(*int*)**, поскольку ***0*** имеет тип ***int***. А для вызова ***print*('*a*')** выбирается ***print*(*char*)**, ибо **'*a*'** имеет тип ***char*** (§4.3.1). Причина раздельного рассмотрения преобразований и продвижений заключается в предпочтении безопасных продвижений, таких как ***char*** в ***int***, небезопасным преобразованиям вроде ***int*** в ***char***.

Результат разрешения перегрузки функций не зависит от порядка объявления функций в программе.

Разрешение перегрузки функций базируется на довольно-таки сложном наборе правил, так что иногда программисту выбор компилятора покажется неочевидным. Здесь уместен вопрос, ради чего все это делается? Рассмотрим альтернативу перегрузки функций. Часто приходится выполнять похожие действия над объектами разных типов. Без перегрузки мы в таких случаях вынуждены определять несколько функций с разными именами:

```
void print_int (int) ;
void print_char (char) ;
void print_string (const char* ) ;

void g (int i, char c, const char* p, double d)
{
  print_int (i) ;      // ok
  print_char (c) ;     // ok
  print_string (p) ;   // ok
  print_int (c) ;      // ok? вызывается print_int(int(c))
  print_char (i) ;     // ok? вызывается print_char(char(i))
  print_string (i) ;   // error
  print_int (d) ;      // ok? вызывается print_int(int(d))
}
```

В итоге нужно помнить все функции и не ошибаться в выборе правильного варианта. Это утомительно, препятствует обобщенному программированию (§2.7.2) и фокусирует внимание на относительно низкоуровневых аспектах типов данных. В отсутствие перегрузки к аргументам вызова функций применяются все стандартные преобразования типов. Это может приводить к ошибкам. Так, в последнем примере из четырех ошибочных вызовов компилятором обнаруживается лишь один. Перегрузка же функций повышает шансы компилятора на то, что неправильные аргументы вызова будут им обнаружены и отвергнуты.

7.4.1. Перегрузка и возвращаемые типы

Возвращаемые типы не участвуют в разрешении перегрузки. Это сделано для обеспечения независимости выбора перегруженных операций (§11.2.1, §11.2.4) и вызова функций от окружающего контекста. Рассмотрим пример:

```
float  sqrt (float) ;
double sqrt (double) ;

void f(double da, float fla)
{
  float fl = sqrt (da) ;      // вызывается sqrt(double)
  double d = sqrt (da) ;      // вызывается sqrt(double)
  fl = sqrt (fla) ;           // вызывается sqrt(float)
  d = sqrt (fla) ;            // вызывается sqrt(float)
}
```

Если возвращаемые типы принимать во внимание, то уже нельзя по одному лишь виду вызова ***sqrt***() решить, о каком варианте функции идет речь.

7.4.2. Перегрузка и области видимости

Перегрузка имен функций имеет место только в *одной и той же области видимости*. Например:

```
void f(int) ;

void g ()
{
  void f(double) ;
  f(1) ;                      // вызывается f(double)
}
```

Очевидно, что ***f(int)*** была бы идеальным соответствием вызову ***f(1)***, но в текущей области видимости объявлен лишь вариант ***f(double)***. В подобных случаях можно вводить и удалять локальные объявления ради достижения желаемого результата. Но как всегда, намеренное сокрытие может быть полезным, а непреднамеренное сокрытие оказаться неприятным сюрпризом. Когда требуется распространить перегрузку поверх классовых областей видимости (§15.2.2) и областей видимости, связанных с пространствами имен (§8.2.9.2), нужно использовать либо *объявления* ***using***, либо *директивы* ***using*** (§8.2.2, §8.2.3). См. также §8.2.6.

7.4.3. Явное разрешение неоднозначностей

Объявление слишком малого (или слишком большого) количества перегруженных вариантов функции может приводить к неоднозначностям. Например:

```
void f1 (char) ;
void f1 (long) ;

void f2 (char*) ;
void f2 (int*) ;
```

```
void k (int i)
{
 f1 (i) ;                // неоднозначность: f1(char) или f1(long)
 f2 (0) ;                // неоднозначность: f2(char*) или f2(int*)
}
```

В таких случаях следует рассмотреть весь набор перегруженных вариантов функции как единое целое и решить, насколько он логичен с точки зрения семантики функции. Часто разрешить неоднозначность можно простым добавлением еще одного варианта функции. Например, добавляя

```
inline void f1 (int n) { f1 (long (n) ) ; }
```

мы добиваемся разрешения неоднозначности *f(i)* в пользу более широкого типа ***long int***.

Для разрешения неоднозначности в конкретном вызове можно воспользоваться явным преобразованием типа. Например:

```
f2 (static_cast<int*> (0) ) ;
```

Однако это не более, чем паллиатив — в каждом новом вызове все надо повторять заново.

Некоторых новичков в программировании на C++ раздражают сообщения компилятора о неоднозначностях. Более опытные программисты ценят эти сообщения об ошибках, ибо видят в них своевременные предупреждения об ошибках в проектировании.

7.4.4. Разрешение в случае нескольких аргументов

В рамках имеющихся правил разрешения перегрузки можно быть уверенным в том, что в случаях, когда точность или эффективность вычислений различаются существенно для разных типов данных, выбран будет самый простой вариант функции (алгоритма вычислений). Например:

```
int pow (int, int) ;
double pow (double, double) ;

complex pow (double, complex) ;
complex pow (complex, int) ;
complex pow (complex, double) ;
complex pow (complex, complex) ;

void k (complex z)
{
  int i = pow (2,2) ;               // вызов pow(int,int)
  double d = pow (2.0,2.0) ;        // вызов pow(double,double)
  complex z22 = pow (2,z) ;         // вызов pow(double,complex)
  complex z33 = pow (z,2) ;         // вызов pow(complex,int)
  complex z44 = pow (z,z) ;         // вызов pow(complex,complex)
}
```

В процессе выбора среди перегруженных функций с двумя и более аргументами на основе правил из §7.4 отбираются функции с наилучшими соответствиями по каждому аргументу. Вызывается в итоге та из них, у которой для одного аргумента

соответствие наилучшее, а требующиеся для других аргументов преобразования не хуже необходимых преобразований у остальных функций. Если такой функции не находится, то вызов отвергается как неоднозначный. Например:

```
void g ()
{
  double d = pow (2.0, 2) ;   // error: pow(int(2.0),2) или pow(2.0,double(2))?
}
```

Здесь вызов неоднозначен, потому что фактический аргумент ***2.0*** наилучшим образом соответствует варианту ***pow*** (***double***, ***double***), а аргумент ***2*** наилучшим образом соответствует ***pow*** (***int***, ***int***).

7.5. Аргументы по умолчанию

Функции общего назначения обычно имеют больше аргументов, чем это необходимо в простых случаях. Например, функции для конструирования классовых объектов (классовые конструкторы — см. §10.2.3) часто для гибкости обеспечивают несколько возможностей. Рассмотрим функцию, предназначенную для печати целого. Решение о предоставление пользователю возможности выбрать основание счисления для выводимых чисел кажется вполне разумным, но в большинстве случаев числа печатаются в десятичном виде. Например, программа

```
void print (int value, int base =10) ;

void f ()
{
  print (31) ;
  print (31, 10) ;
  print (31, 16) ;
  print (31, 2) ;
}
```

выводит следующую последовательность:

31 31 1f 11111

Эффекта от применения аргумента по умолчанию можно добиться и перегрузкой:

```
void print (int value, int base) ;
inline void print (int value) {print (value, 10) ; }
```

Однако перегрузка менее четко отражает тот факт, что нужна-то единственная функция печати плюс укороченный вариант ее вызова для наиболее типичного случая.

Тип умолчательного аргумента проверяется в месте объявлении функции и вычисляется в месте ее вызова. Умолчательными значениями могут снабжаться только аргументы, стоящие в *самом конце списка аргументов*. Например:

```
int f (int, int=0, char*=0) ;   // ok
int g (int=0, int=0, char*) ;   // error
int h (int=0, int, char*=0) ;   // error
```

Обратите внимание на то, что пробел между * и = обязателен (*= означает операцию присваивания; §6.2):

```
int nasty(char*=0);          // синтаксическая ошибка
```

Аргумент по умолчанию нельзя ни повторять, ни изменять в той же самой области видимости. Например:

```
void f(int x=7);
void f(int=7);               // error: нельзя повторять аргумент по умолчанию
void f(int=8);               // error: другое значение умолчательного аргумента

void g()
{
  void f(int x=9);           // ok: это объявление скрывает внешние объявления
  // ...
}
```

Объявление имени во вложенной области видимости, скрывающее объявление того же имени в объемлющей области видимости, чревато ошибками.

7.6. Неуказанное число аргументов

Для некоторых функций бывает невозможно заранее указать количество и типы всех аргументов вызова. Объявления таких функций содержат список аргументов, завершающийся *многоточием* (*ellipsis*, ...), означающим, что «могут быть еще аргументы». Например:

```
int printf(const char* ...);
```

Это объявление означает, что при вызове стандартной библиотечной функции ***printf***() (§21.8) должен быть указан хотя бы один фактический аргумент типа ***char****, но также могут быть (а могут и не быть) и иные аргументы. Например:

```
printf("Hello, world!\n");
printf("My name is %s %s\n", first_name, second_name);
printf("%d + %d=%d\n",2,3,5);
```

В процессе интерпретации списка фактических аргументов вызова такие функции должны опираться на информацию, недоступную компилятору. Для функции ***printf***() первым аргументом служит так называемая *управляющая строка* (*format string*), содержащая специальные последовательности символов, которые и позволяют функции ***printf***() корректно обрабатывать последующие аргументы; %***s*** означает «ожидается аргумент типа ***char****», а %***d*** означает, что «ожидается аргумент типа int». Компилятор, в общем случае, этого знать не может и не может гарантировать, что ожидаемые дополнительные аргументы будут на месте, или что их тип будет корректным. Например, программа

```
#include <stdio.h>

int main()
{
  printf("My name is %s %s\n",2);
}
```

скомпилируется и (в лучшем случае) выдаст что-нибудь странное (попробуйте на практике!).

Ясно, что если аргумент не был объявлен, то компилятор не имеет необходимой информации для проверки типов фактических аргументов и их стандартных преобразований. В таких случаях ***char*** или ***short*** передаются как ***int***, а ***float*** передается как ***double***. Не факт, что программист ожидает именно этого.

Хорошо спроектированная программа нуждается лишь в минимальном количестве функций, типы аргументов которых определены не полностью. Чаще всего, вместо функций с неопределенными аргументами можно использовать перегрузку функций и аргументы по умолчанию, что гарантирует надежную проверку типов. Функциями с многоточием следует пользоваться лишь тогда, когда и количество аргументов не определено, и их типы неизвестны. Типичнейшим примером таких функций служат функции библиотеки языка C, разработанные до того, как появились их альтернативы на языке C++:

```
int fprintf(FILE*, const char* ...);    // из <cstdio>
int execl(const char* ...);             // из UNIX заголовоч. файла
```

Набор стандартных макросов для доступа к неспецифицированным аргументам этих функций определен в файле <***cstdarg***>. Напишем функцию ***error***() для вывода сообщений об ошибках. У этой функции неопределенное число аргументов, из которых первый имеет тип ***int*** и означает степень серьезности ошибки, а остальные аргументы — строки. Основная идея состоит в том, чтобы составлять сообщение об ошибке из отдельных слов, передаваемых с помощью строковых аргументов. Список строковых аргументов должен оканчиваться нулевым указателем на ***char***:

```
extern void error(int...);
extern char* itoa(int, char[]);        // см. §6.6[17]

const char* Null_cp=0;

int main(int argc, char* argv[])
{
  switch(argc)
  {
    case 1:
      error(0,argv[0],Null_cp);
      break;
    case 2:
      error(0,argv[0],argv[1],Null_cp);
      break;
    default:
      char buffer[8];
      error(1,argv[0], "with",itoa(argc-1,buffer),"arguments", Null_cp);
  }
    //...
}
```

Функция ***itoa***() возвращает строковое представление для своего целого аргумента.

Отметим, что использование целого значения ***0*** для терминирования списка строк не переносимо, ибо на разных системах целочисленный нуль и нулевой ука-

затель могут иметь разные представления. Это иллюстрирует те дополнительные тонкости и сложности, с которыми приходится иметь дело программисту, как только из-за применения многоточия пропадает проверка типов компилятором.

Определим функцию вывода сообщений об ошибках следующим образом:

```
void error (int severity ...)  // За "severity" следует список элементов
                               // типа char* с терминальным нулем
{
  va_list ap;
  va_start (ap, severity) ;

  for (;;)
  {
    char* p = va_arg (ap, char*) ;
    if (p == 0) break;
    cerr << p << ' ';
  }

  va_end (ap) ;

  cerr << '\n' ;
  if (severity) exit (severity) ;
}
```

Сначала создается переменная типа ***va_list*** и инициализируется вызовом ***va_start***(). *Макрос* ***va_start*** в качестве аргументов берет имя указанной переменной и имя последнего формального параметра нашей функции ***error***(). *Макрос* ***va_arg***() последовательно извлекает неименованные фактические аргументы. При каждом вызове программист должен указывать ожидаемый тип; макрос ***va_arg***() полагает, что тип переданного аргумента в точности соответствует указанному программистом, но не может это проверить. Перед выходом из функции, использовавшей ***va_start***(), нужно вызвать ***va_end***(), так как ***va_start***() может модифицировать стек таким образом, что нормальный выход из функции становится невозможным. Вызов ***va_end***() устраняет эти модификации.

7.7. Указатели на функции

С функциями можно выполнить лишь два вида работ: *вызвать* или *вычислить их адрес*. Указатель, которому присвоен адрес функции, можно далее использовать для вызова этой функции. Например:

```
void error (string s) { /* ... */ }
void (*efct) (string) ;         // указатель на функцию

void f()
{
  efct=&error;                  // efct указывает на функцию error
  efct ("error") ;              // вызов error через efct
}
```

Компилятор легко обнаруживает, что ***efct*** является указателем на функцию, и вызывает функцию по адресу, содержащемуся в этом указателе. При этом *разыменование*

указателя на функцию (то есть применение операции *) *не обязательно.* Также не обязательно использовать явным образом операцию & для получения адреса функции:

```
void (*f1) (string) = &error;  // ok
void (*f2) (string) = error;   // ok; тот же смысл, что и &error

void g ()
{
 f1 ("Vasa") ;                 // ok
  (*f1) ("Mary Rose") ;        // тоже ok
}
```

Для указателей на функции надо объявлять типы аргументов так же, как и для самой функции. Присваивание указателей на функции друг другу допустимо лишь при полном совпадении типов функций. Например:

```
void  (*pf) (string) ;         // указатель на void(string)
void f1 (string) ;             // void(string)
int f2 (string) ;              // int(string)
void f3 (int* ) ;              // void(int*)

void f()
{
 Pf = &ff1;                    // ok
 pf = &ff2;                    // error: не тот возвращаемый тип
 pf = &ff3;                    // error: не тот тип аргумента

 Pf("Hera") ;                  // ok
 pf(1) ;                       // error: не тот тип аргумента

 int i=p ("Zeus") ;            // error: void присваивается переменной типа int
}
```

Правила передачи аргументов при вызове функций через указатели те же самые, что и при непосредственном вызове функций.

Часто бывает удобным один раз определить синонимичное имя для типов указателей на функции, чтобы избежать многократного использования достаточно неочевидного синтаксиса их объявления. Вот пример из заголовочного файла системы UNIX:

```
typedef void  (*SIG_TYP) (int) ;       // из <signal.h>
typedef void  (*SIG_ARG_TYP) (int ) ;
SIG_TYP signal (int, SIG_ARG_TYP) ;
```

Часто бывают полезными массивы указателей на функции. Например, система меню в моем графическом редакторе, реализована через массивы указателей на функции, каждая из которых выполняет соответствующее действие. В целом это решение нельзя здесь рассмотреть в деталях, но вот его основная идея:

```
typedef void  (*PF) () ;

// команды редактирования:
PF edit_ops [ ] = { &cut, &paste, &copy, &search } ;

// файловые операции:
PF file_ops [ ] = { &open, &append, &close, &write } ;
```

Теперь можно определять и инициализировать указатели, ответственные за выполнение действий, инициируемых при помощи меню и кнопок мыши:

```
PF* button2 = edit_ops;
PF* button3 = file_ops;
```

В полной реализации с каждым пунктом меню нужно связывать больше информации. Например, нужно где-то хранить строку с текстом данного пункта меню. Во время работы роль кнопок мыши может изменяться в зависимости от контекста. Такие изменения реализуются (частично) изменением значений указателей, относящихся к кнопкам. При выборе конкретной кнопкой (например, кнопкой 2) некоторого пункта меню (например, пункта 3) выполняется соответствующая операция:

```
button2[2]();                   // вызов 3-ей функции из button2
```

Чтобы по достоинству оценить всю мощь указателей на функции, нужно попробовать написать подобный код без их помощи, и без помощи их еще более полезных родственников — виртуальных функций (§12.2.6). Меню можно модифицировать во время выполнения программы, просто добавляя новые функции (обработчики) в таблицу операций данного меню. Также легко создавать новые меню динамически (во время выполнения программы).

Указатели на функции можно применить для реализации простой формы *полиморфных процедур*, то есть процедур, *применимых к объектам разных типов*:

```
typedef int (*CFT) (const void*, const void*);

void ssort (void* base, size_t n, size_t sz, CFT cmp)
/*
  Сортировка n элементов вектора base в возрастающем порядке с использованием
  функции сравнения, указуемой с помощью cmp.
  Элементы имеют размер sz.
  Shell sort (Knuth, Vol3, pg84)
*/
{
 for (int gap=n/2; 0<gap; gap/=2)
  for (int i=gap; i<n; i++)
   for (int j=i-gap; 0<=j; j-=gap)
   {
    char* b=static_cast<char*>(base);   // обязательное приведение типа
    char* pj=b+j*sz;                     // &base[j]
    char* pig=b+(j+gap)*sz               // &base[j+gap]

    if (cmp(pjg,pj)<0)                   // обменять base[j] и base[j+gap];
    {
      for (int k=0; k<sz; k++)
      {
        char temp =pj[k];
        pj[k] = pjg[k];
        pjg[k]= temp;
      }
    }
    else
```

```
            break;
        }
    }
```

Процедура *ssort*() не знает типы объектов, которые она сортирует, а только число элементов (размер массива), размер каждого элемента и функцию, используемую для сравнения элементов. Тип процедуры *ssort*() намеренно выбран таким же, как у стандартной сортирующей процедуры *qsort*() из библиотеки языка С. В реальных программах применяются *qsort*(), алгоритм *sort*() стандартной библиотеки С++ (§18.7.1), или специализированные процедуры сортировки. Рассмотренный стиль кодирования типичен для языка С, но его нельзя назвать наиболее подходящим способом формулирования алгоритмов на языке С++ (§13.3, §13.5.2).

Разработанная нами процедура *ssort*() применима для сортировки табличной информации:

```
struct User
{
  char* name;
  char* id;
  int   dept;
};

User heads[]={ "Ritchie D.M",    "dmr",   11271,
               "Sethi R.",       "ravi",  11272,
               "Szymanski T.G.", "tgs",   11273,
               "Schryer N.L.",   "nls",   11274,
               "Schryer N.L.",   "nls",   11275,
               "Kernighan B.W.", "bwk",   11276 };

void print_id(User* v, int n)
{
  for(int i=0; i<n; i++)
    cout<< v[i].name<< '\t'<< v[i].id<< '\t' <<v[i].dept<< '\n';
}
```

Нужно еще определить подходящую функцию сравнения. Такая функция должна возвращать отрицательное значение, когда первый аргумент меньше второго, нуль в случае равенства аргументов, и положительное число в остальных случаях:

```
int cmp1(const void* p, const void* q)        // сравнение имен
{
  return strcmp( static_cast<const User*>(p)->name,
                 static_cast<const User*>(q)->name );
}

int cmp2(const void* p, const void* q)        // сравнение номеров отделов
{
  return  static_cast<const User*>(p)->dept -
          static_cast<const User*>(q)->dept;
}
```

Саму сортировку и вывод ее результатов выполним в функции *main*():

```
int main()
{
  cout<<"Heads in alphabetical order:\n";
```

```
    ssort(heads,6,sizeof(User),cmp1);
    print_id(heads,6);
    cout<< '\n';

    cout<< "Heads in order of department number:\n";
    ssort(heads,6,sizeof(User),cmp2);
    print_id(heads,6);
}
```

Указателям на функции можно присваивать адреса перегруженных функций. В таких случаях тип указателя используется для выбора нужного варианта перегруженной функции. Например:

```
void f(int);
int f(char);

void (*pf1)(int)=&f;        // void f(int)
int(*pf2)(char)=&f;         // int f(char)
void (*pf3)(char)=&f;       // error: нет функции void f(char)
```

Функции вызываются через указатели на функции исключительно с правильными типами аргументов и возвращаемых значений. Когда производится присваивание указателям на функции или выполняется их инициализация, *никаких неявных преобразований типов аргументов или типов возвращаемых значений не производится.* Это означает, что функция

```
int cmp3(const mytype*,const mytype*);
```

не подходит в качестве аргумента для *ssort*(). Причина в том, что в противном случае была бы нарушена гарантия вызова *cmp3*() с аргументами типа const *mytype** (см. также §9.2.5).

7.8. Макросы

Макросы очень полезны в языке С, но в языке С++ они используются гораздо реже. Самое первое правило для макросов: не используйте их без крайней необходимости. Почти что каждый макрос свидетельствует о наличии слабых мест в языке, программе или программисте. Так как из-за макросов текст программы изменяется до того, как его увидит компилятор, то создаются лишние проблемы для многих инструментов программирвания. Если вы применяете макросы, приготовьтесь получать меньшую пользу от отладчиков, генераторов перекрестных ссылок и профилировщиков. Если все же вам нужно применять макросы, внимательно прочитайте руководство по вашей реализации препроцессора С++ и не прибегайте к уж слишком хитрым конструкциям. Также следуйте общепринятому соглашению об именовании макросов с помощью исключительно заглавных букв. Синтаксис макросов описан в §А.11.

Простейший макрос определяется следующим образом:

```
#define NAME rest of line
```

Всюду, где встречается лексема *NAME*, она заменяется на *rest of line*. Например:

```
named = NAME
```

превратится в

```
named = rest of line
```

Можно определять макросы с аргументами. Например:

```
#define MAC(x,y) argument1: x argument2: y
```

В местах, где применяется макрос *MAC*, должны также присутствовать и два строковых аргумента. Они заменят *x* и *y* при макроподстановке. Например,

```
expanded = MAC(foo bar, yuk yuk)
```

превратится в

```
expanded = argument1: foo bar argument2: yuk yuk
```

Имена макросов *перегружать нельзя*. Также нельзя использовать в них *рекурсивные вызовы* (препроцессор с ними не справится):

```
#define PRINT(a,b) cout<<(a)<<(b)
#define PRINT(a,b,c) cout<<(a)<<(b)<<(c) /*беда?: нет перегрузки, переопределение*/

#define FAC(n) (n>1)?n*FAC(n-1):1          /*беда: рекурсивное макро*/
```

Макросы связаны лишь с текстовой обработкой и они мало что знают о синтаксисе языка C++, и вообще ничего не знают о его типах и областях видимости. Компилятор видит программный текст после макроподстановки, так что ошибки в макросах обнаруживаются лишь как последствия, а вовсе не в определениях макросов. Все это приводит к весьма туманным сообщениям об ошибках.

Вот макросы, внушающие доверие:

```
#define CASE break;case
#define FOREVER for(;;)
```

А вот примеры совершенно ненужных макросов:

```
#define PI 3.141593
#define BEGIN {
#define END }
```

Следующие макросы опасны:

```
#define SQUARE(a) a*a
#define INCR_xx (xx)++
```

Чтобы убедиться в их опасности, попробуйте применить их:

```
int xx = 0;                    // глобальный счетчик

void f()
{
  int xx = 0;                  // локальная переменная
  int y = SQUARE(xx+2);        // y=xx+2*xx+2; то есть y=xx+(2*xx)+2
  INCR_xx;                     // инкрементирует локальную xx
}
```

Если уж вам действительно нужны макросы, применяйте операцию разрешения области видимости :: при ссылке на глобальные имена (§4.9.4) и везде, где только можно, заключайте в круглые скобки имена их аргументов. Например:

```
#define MIN(a,b) (((a)<(b))?(a):(b))
```

Если вы написали довольно сложный макрос, так что требуются комментарии к нему, применяйте комментарии вида /* */, ибо часто в состав инструментов программирования на C++ входит препроцессор языка C, а он ничего не знает о комментариях вида //. Например:

```
#define M2(a) something(a)    /* полезный комментарий */
```

При помощи макросов вы можете создать свой собственный язык. Даже если вы сами предпочтете такой «улучшенный» язык простому C++, другим программистам он будет непонятен. Более того, препроцессор C — это очень простой макропроцессор. Поэтому, когда вы захотите создать что-нибудь нетривиальное, то окажется, что либо это невозможно, либо неоправданно трудоемко. Механизмы ***const***, ***inline***, ***template***, ***enum*** и ***namespace*** являются альтернативой традиционному использованию препроцессорных конструкций. Например:

```
const int answer = 42;
template<class T> inline T min(T a, T b) {return (a<b)?a:b;}
```

При помощи макросов можно создавать новые имена — новую строку можно составить из двух строк при помощи *операции препроцессора* ##. Например:

```
#define NAME2(a,b) a##b
int NAME2(hack,cah)();
```

превратится в

```
int hackcah();
```

что и достанется компилятору для чтения.

Директива препроцессора

```
#undef X
```

гарантирует, что более не существует макроса с именем *X* независимо от того, существовал ли он до этой директивы, или нет. Такой прием позволяет на всякий случай защититься от нежелательных макросов, так как в точности узнать их действие на фрагмент кода бывает нелегко.

7.8.1. Условная компиляция

Одного случая применения макросов избежать практически невозможно. Директива компилятора #***ifdef identifier*** заставляет опустить последующую часть кода, пока не встретится директива #***endif***. Например:

```
int f(int a
#ifdef arg_two
,int b
#endif
);
```

превратится для компилятора в

```
int f(int a
);
```

если не определен макрос ***arg_two***. Данный код способен запутать автоматические инструменты разработки, так как они рассчитывают на разумное поведение программиста.

Но в большинстве случаев характер применения *#ifdef* менее эксцентричный, чем в рассмотренном примере, и при наличии определенных ограничений эта директива приносит мало вреда. См. также §9.3.3.

Следует аккуратно выбирать имена макросов, используемых в директиве *#ifdef*, чтобы они не вступали в противоречие с обычными идентификаторами. Например:

```
struct Call_info
{
 Node* arg_one;
 Node* arg_two;
 // ...
};
```

Этот невинно выглядящий код вызовет недоразумения, как только кто-нибудь определит следующий макрос:

```
#define arg_two x
```

К сожалению, многие стандартные заголовочные файлы содержат множество ненужных и опасных макросов.

7.9. Советы

1. Относитесь с подозрением к не***константным*** аргументам, передаваемым по ссылке; если вы хотите, чтобы функция модифицировала аргументы, используйте указатели и возвращаемое значение; §5.5.
2. Применяйте ***константные*** ссылки, если вы хотите минимизировать копирование аргументов; §5.5.
3. Используйте ***const*** активно и последовательно; §7.2.
4. Избегайте макросов; §7.8.
5. Избегайте функций с неуказанным числом аргументов; §7.6.
6. Не возвращайте указатели или ссылки на локальные переменные; §7.3.
7. Применяйте перегрузку функций в случаях, когда концептуально одинаковая работа выполняется над данными разных типов; §7.4.
8. В случае перегрузки функций с целыми аргументами реализуйте полный набор вариантов для устранения неоднозначностей; §7.4.3.
9. Обдумывая вопрос о применении указателей на функций, рассмотрите возможность их замены на виртуальные функции (§2.5.5) или шаблоны (§2.7.2) в качестве лучших альтернатив; §7.7.
10. Если вам нужны макросы, применяйте для них безобразно выглядящие имена из заглавных букв; §7.8.

7.10. Упражнения

1. (*1) Напишите следующие объявления: функция с аргументами «указатель на символ» и «ссылка на целое», не имеющая возврата; указатель на такую функцию; функция с таким указателем в качестве аргумента; функция, возвращающая такой указатель. Напишите определение функции, принимающей такой указатель в качестве аргумента и возвращающей его же. Подсказка: воспользуйтесь ***typedef***.
2. (*1) Что означает следующее? Для чего это может потребоваться?

```
typedef int (&rifii) (int, int) ;
```

3. (*1.5) Напишите программу типа «Hello, world!», которая берет имя (*name*) из командной строки и выводит «Hello, *name*!». Модифицируйте программу так, чтобы она могла брать произвольное количество имен в качестве аргументов и приветствовать всех по этим именам.
4. (*1.5) Напишите программу, которая читает произвольное количество файлов, чьи имена задаются в командной строке, и выводит их последовательно в ***cout***. Поскольку эта программа соединяет содержимое своих аргументов для формирования вывода, можете назвать ее ***cat***.
5. (*2) Преобразуйте небольшую программу на языке С в соответствующую программу на С++. Переделайте заголовочные файлы таким образом, чтобы в них объявлялись все вызываемые в программе функции, и чтобы были указаны типы всех их аргументов. Где можно, замените все директивы #***define*** на ***enum***, ***const*** или ***inline***. Удалите все объявления ***extern*** из .***c***-файлов и в случае необходимости преобразуйте определения функций в синтаксисе языка С в соответствующие определения в синтаксисе языка С++. Замените вызовы функций ***malloc***() и ***free***() на операции ***new*** и ***delete***. Удалите явные преобразования типов, в которых нет необходимости.
6. (*2) Реализуйте функцию ***ssort***() (§7.7) с помощью более эффективного алгоритма сортировки. Подсказка: ***qsort***().
7. (*2.5) Имеется структура ***Tnode***:

```
struct Tnode
{
  string word;
  int count;
  Tnode* left;
  Tnode* right ;
};
```

Напишите функцию для внедрения новых слов в дерево с узлами типа ***Tnode***. Напишите функцию для вывода такого дерева. Напишите функцию для вывода слов из такого дерева в алфавитном порядке. Модифицируйте структуру ***Tnode*** так, чтобы она содержала указатель на произвольно длинное слово, хранящееся в массиве символов, память под который выделяется операцией new. Модифицируйте все функции, чтобы они работали с новым вариантом структуры ***Tnode***.

8. (*2.5) Напишите функцию, инвертирующую (транспонирующую) двумерный массив. Подсказка: §C.7.
9. (*2) Напишите программу шифрования, которая читает из ***cin*** и пишет закодированные символы в ***cout***. Можете применить следующую простую схему шифрования: символ c кодируется выражением ***c^key[i]***, где ***key*** — строка, передаваемая в качестве аргумента командной строки. Программа циклически использует символы из строки ***key*** до исчерпания ввода. Повторное шифрование по этой же формуле восстанавливает исходный текст. Если нет входного текста или ***key*** есть нулевая строка, шифрование не выполняется.
10. (*3.5) Напишите программу, помогающую без знания ключа дешифровать текст, закодированный программой из предыдущего упражнения. Подсказка: David Kahn: *The Codebreakers*, Macmillan, 1967, New York, pp. 207-213.
11. (*3) Напишите функцию ***error()***, принимающую строку в стиле функции ***printf()***, содержащую ***%s***, ***%c*** и ***%d***, и произвольное число других аргументов. Не используйте функцию ***printf()***. См. §21.8. Используйте <***cstdarg***>.
12. (*1) Как бы вы выбирали имена для типов указателей на функции, которые определяются с помощью ***typedef***?
13. (*2) Просмотрите несколько программ, обращая внимание на множество стилей именования. Как используются заглавные буквы? Как используется символ подчеркивания? Когда используются короткие имена вроде ***i*** или ***x***?
14. (*1) Что плохого в следующих макросах?

```
#define PI = 3.141593;
#define MAX(a,b) a>b?a:b
#define fac(a) (a)*fac((a)-1)
```

15. (*3) Напишите простой макропроцессор, позволяющий определять и расширять макросы (как это делает препроцессор языка С). Читайте из ***cin*** и выводите в ***cout***. Вначале ограничьтесь макросами без аргументов. Подсказка: программа-калькулятор (§6.1) содержит таблицу символов и лексический анализатор, которыми вы можете воспользоваться.
16. (*2) Реализуйте функцию ***print()*** из §7.5.
17. (*2) Добавьте функции, такие как ***sqrt()***, ***log()*** и ***sin()***, к программе-калькулятору из §6.1. Подсказка: заранее определите эти имена и вызывайте функции с помощью массива указателей на функции. Не забывайте проверять фактические аргументы вызова.
18. (*1) Напишите функцию вычисления факториала, не использующую рекурсию. См. §11.14[6].
19. (*2) Напишите функции, которые добавляют день, месяц, год к заданной дате (структура ***Date*** из §5.9[13]). Напишите функцию, вычисляющую день недели для заданного значения ***Date***. Напишите функцию, вычисляющую значение ***Date*** для первого понедельника после заданного ***Date***.

8

Пространства имен и исключения

Год 787! От Рождества Христова?
— Монти Пайтон

Нет столь общих правил, что не допускали бы исключений.
— Роберт Бартон

Модульность — интерфейсы и исключения — пространства имен — ***using*** — ***using namespace*** — разрешение конфликтов имен — поиск имен — композиция пространств имен — псевдонимы пространств имен — пространства имен и код на С — исключения — ***throw*** и ***catch*** — исключения и структура программы — советы — упражнения.

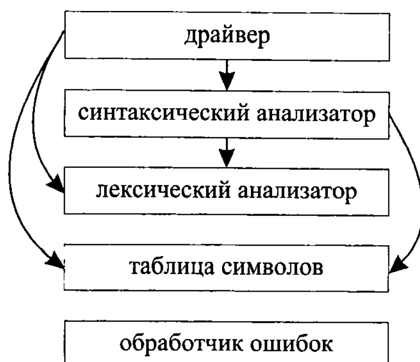
8.1. Разбиение на модули и интерфейсы.

Любая реальная программа состоит из нескольких отдельных частей. Например, даже столь простая программа, как «Hello, world!», включает по крайней мере две части: пользовательский код, требующий выполнить вывод строки ***Hello, world!***, и система ввода-вывода, которая и осуществляет этот вывод.

Снова рассмотрим программу-калькулятор из §6.1. Ее можно рассматривать как совокупность пяти частей:

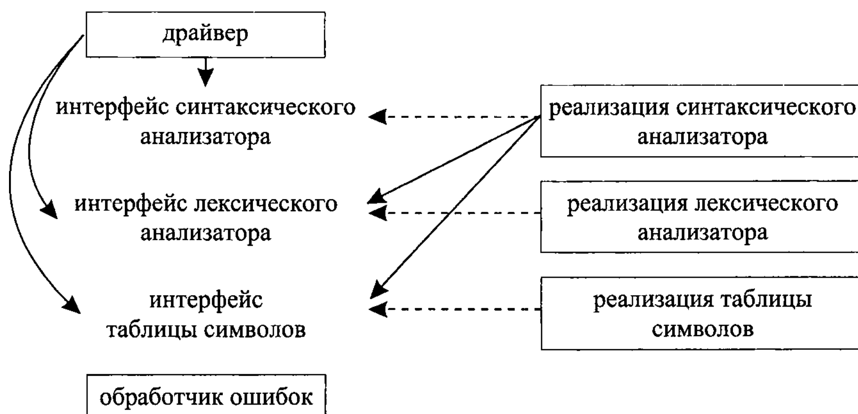
1. Парсера, выполняющего синтаксический анализ
2. Лексического анализатора, составляющего лексемы из символов
3. Таблицы символов, содержащей пары (строка,значение)
4. Управляющей части, содержащей функцию ***main***()
5. Обработчика ошибок

Это можно отобразить графически:



Здесь стрелками обозначено отношение «использует». Чтобы упростить рисунок, я не стал на нем отображать тот факт, что все части используют обработчик ошибок. По сути, калькулятор состоит лишь из трех частей, а управляющая часть («драйвер») и обработчик ошибок введены в него для полноты реализации.

Когда один модуль использует другой модуль, ему нет никакой необходимости знать все детали устройства последнего. В идеале, большая часть внутреннего устройства модуля неизвестна его клиентам. Мы проводим четкое различие между устройством модуля и его интерфейсом. Например, парсер обращается напрямую лишь к интерфейсу лексического анализатора. А лексический анализатор реализует все анонсируемые им посредством интерфейса сервисы. Это можно изобразить графически:



Пунктирные линии означают «реализует». Я считаю, что эта схема отражает реальную структуру программы, а задача программиста сводится к тому, чтобы точно отразить ее в коде. В этом случае код будет простым, эффективным, понятным, удобным для сопровождения и т.д., потому что он напрямую соответствует фундаментальным основам нашего проекта.

Последующие разделы данной главы показывают, как сделать простой и понятной логическую структуру программы калькулятора, а в §9.3 рассматривается вопрос, как физически организовать исходный текст программы, чтобы он наилучшим образом соответствовал этой структуре. Калькулятор — это крошечная программа, для которой в реальных условиях я не стал бы столь интенсивно использовать пространства имен и раздельную компиляцию (§2.4.1, §9.1). Все это используется для демонстрации приемов, позволяющих писать большие программы так, чтобы потом не утонуть в них. В реальных программах каждый модуль, представленный отдельным пространством имен, часто содержит сотни функций, классов, шаблонов и т.п.

Ради демонстрации различных практических приемов и языковых средств я намеренно провожу разбиение программы калькулятора на модули в несколько этапов. В «реальной жизни» программы вряд ли разрабатываются в такой последовательности. Опытный программист может сразу начать с почти готового проектного решения. Тем не менее, в процессе многолетней эксплуатации программы и ее модернизации не исключена и кардинальная переделка ее структуры.

Обработка ошибок проходит сквозной линией через всю структуру программы. Разбивая программу на модули, или (наоборот) собирая ее из модулей, мы должны заботиться о минимизации зависимости между модулями, проистекающей из обработки ошибок. В языке C++ обнаружение ошибок (сообщение о них) и их обработку можно четко разделить с помощью механизма исключений. Поэтому рассмотрение пространств имен в качестве модулей (§8.2) дополняется рассмотрением исключений, помогающим еще более улучшить модульность программы (§8.3).

В данной и следующей главах рассматривается лишь небольшая часть из всего множества вопросов, связанных с модульностью программ. Некоторые аспекты модульности можно было бы дополнительно рассмотреть на примере параллельно работающих и взаимодействующих между собой процессов. Модульность можно рассматривать и в смысле раздельных адресных пространств, между которыми осуществляется передача данных. Все эти аспекты модульности довольно независимы и ортогональны друг другу. Самое интересное, что разбиение на модули почти всегда выполняется довольно просто. Намного сложнее — обеспечить безопасное, удобное и эффективное взаимодействие модулей между собой.

8.2. Пространства имен

Пространства имен являются механизмом логического группирования программных объектов. Если некоторые объявления согласно определенному критерию логически близки друг другу, то для отражения этого факта их можно поместить в одно и то же пространство имен. Например, все объявления из программы калькулятора (§6.1.1), относящиеся к синтаксическому анализатору (парсеру), можно поместить в одно пространство имен ***Parser***:

```
namespace Parser
{
  double expr(bool) ;
  double prim(bool get) { /* ... */ }
  double term(bool get) { /* ... */ }
  double expr(bool get) { /* ... */ }
}
```

Функция *expr*() должна быть сначала объявлена, а определена потом из-за необходимости как-то разорвать порочный круг зависимостей, рассмотренный §6.1.1.

Часть программы калькулятора, отвечающая за обработку ввода, также может быть помещена в свое собственное пространство имен:

```
namespace Lexer
{
  enum Token_value
  {
    NAME,        NUMBER,       END,
    PLUS='+',    MINUS='-',    MUL='*',    DIV='/',
    PRINT=';',   ASSIGN='=',   LP='(',     RP=')'
  };

  Token_value curr_tok;
  double number_value;
  string string_value;
  Token_value get_token() { /* ... */ }
}
```

Применение пространств имен наглядно показывает, что именно лексический и синтаксический анализаторы предлагают своим пользователям. Однако если бы я включил в них полные определения функций, то смысл пространств имен уже не проступал бы столь отчетливо. Когда в пространства имен реального размера включают тела функций, приходится просматривать множество страниц кода (или пролистывать множество экранов монитора), чтобы узнать, какие же собственно предоставляются сервисы, то есть каков интерфейс модуля.

Альтернативой раздельного определения интерфейсов является применение программных средств, автоматически извлекающих интерфейсы из модуля реализации. Я не считаю это хорошим решением, потому что: определение интерфейсов является существенной частью процесса проектирования (§23.4.3.4), модуль может предоставлять разные интерфейсы разным пользователям, и к тому же разработка интерфейсов часто ведется задолго до конкретизации деталей реализации.

Вот новый вариант пространства имен *Parser*, в котором интерфейс явным образом отделен от реализации:

```
namespace Parser
{
  double prim(bool);
  double term(bool);
  double expr(bool);
}

double Parser::prim(bool get) { /* ... */ }
double Parser::term(bool get) { /* ... */ }
double Parser::expr(bool get) { /* ... */ }
```

Обратите внимание на то, что теперь каждая функция имеет ровно одно объявление, и одно определение. Пользователям нужно ознакомиться с интерфейсом, содержащим лишь объявления. А реализация может быть помещена в какое-нибудь другое место, куда пользователю заглядывать нет необходимости.

Как продемонстрировано в последнем примере, элемент (*member-name*) пространства имен (*namespace-name*) может быть в нем лишь объявлен, а определен позднее в нотации *namespace-name::member-name*.

Члены (элементы) пространства имен объявляются следующим образом:

```
namespace namespace-name
{
  // объявления и определения
}
```

Нельзя объявлять новый элемент пространства имен вне этого определения (не поможет и явная квалификация имени):

```
void Parser::logical(bool);        // error: нет никакого logical() в Parser
```

Идея состоит в том, чтобы все элементы пространства имен легко обнаруживались в одном определении, и чтобы можно было легко отлавливать ошибки, связанные, например, с описками или несоответствием типов. Например:

```
double Parser::trem(bool);         // error: нет никакого trem() в Parser
double Parser::prim(int);          // error: Parser::prim() принимает аргумент bool
```

Пространство имен формирует свою собственную область видимости. Таким образом, пространство имен является одновременно и фундаментальной, и достаточно простой концепцией. Чем больше размер программы, тем большую пользу приносят пространства имен, помогая четко разделять программу на логические части. Обычные локальные и глобальные области видимости, а также классы, формируют свои пространства имен (§C.10.3).

В идеале, каждая программная сущность должна принадлежать некоторой четко ограниченной логической единице («модулю»). Поэтому любое объявление в нетривиальной программе должно в идеале помещаться в пространство имен, собственное имя которого должно отражать его логическую роль в программе. Исключение составляет функция **main**(), которая должна оставаться глобальной, чтобы исполнительная система всегда могла отыскать стартовую функцию (§8.3.3).

8.2.1. Квалифицированные имена

Пространство имен является отдельной областью видимости. Обычные правила для областей видимости распространяются и на пространства имен, так что если имя объявлено ранее в том же самом пространстве имен (или в охватывающей области видимости), то его можно использовать без проблем. Имена же из других пространств имен требуют дополнительной квалификации. Например:

```
double Parser::term(bool get)      // нужна квалификация Parser::
{
  double left = prim(get);         // нет нужды в квалификации

  for(;;)
    switch (Lexer::curr_tok)       // нужна квалификация Lexer::
    {
      case Lexer::MUL:             // нужна квалификация Lexer::
```

```
        left*=prim(true);          // нет нужды в квалификации
      // ...
    }
    // ...
}
```

Квалификатор *Parser* указывает, что функция *term()* объявлена именно в пространстве имен *Parser*, а не где-то еще (например, в глобальной области видимости). Так как *term()* является членом *Parser*, нет необходимости квалифицировать иной член этого же пространства имен — *prim()*. А вот если убрать квалификатор *Lexer*, то имя *curr_tok* будет считаться необъявленным, поскольку имена из пространства имен *Lexer* не входят в область видимости пространства имен *Parser*.

8.2.2. Объявления using

Когда имя часто используется вне пределов своего пространства имен, может быть утомительно то и дело дополнительно квалифицировать его. Рассмотрим следующий пример:

```
double Parser::prim(bool get)     // обработка первичных выражений
{
  if(get) Lexer::get_token();

  switch(Lexer::curr_tok)
  {
    case Lexer::NUMBER:           // константа с плавающей запятой
      Lexer::get_token();
      return Lexer::number_value;
    case Lexer::NAME:
    {
      double& v=table[Lexer::string_value];
      if(Lexer::get_token()==Lexer::ASSIGN) v=expr(true);
      return v;
    }
    case Lexer::MINUS:            // унарный минус
      return -prim(true);
    case Lexer::LP:
    {
      double e = expr(true);
      if(Lexer::curr_tok!= Lexer::RP) return Error::error(") expected");
      Lexer::get_token();         // пропустить скобку ')'
      return e;
    }
    case Lexer::END:
      return 1;
    default:
      return Error::error("primary expected");
  }
}
```

Назойливое повторение квалификатора *Lexer* утомительно и отвлекает внимание. Такую избыточность можно устранить с помощью *объявления using* (*using-decla-*

ration), которое позволяет однократно указать для текущей области видимости, что имя ***get_token*** взято из пространства имен ***Lexer***. Например:

```
double Parser::prim(bool get)    // обработка первичных выражений
{
 using Lexer::get_token;         // использовать get_token из Lexer
 using Lexer::curr_tok;          // использовать curr_tok из Lexer
 using Error::error;             // использовать error из Error

 if(get) get_token();

 switch(curr_tok)
 {
   case Lexer::NUMBER:           // константа с плавающей запятой
     get_token();
     return Lexer::number_value;
   case Lexer::NAME:
   {
     double& v = table[Lexer::string_value];
     if(get_token() == Lexer::ASSIGN) v = expr(true);
     return v;
   }
   case Lexer::MINUS:            // унарный минус
     return -prim(true);
   case Lexer::LP:
   {
     double e=expr(true);
     if(curr_tok != Lexer::RP) return error(") expected");
     get_token();                // пропустить скобку ')'
     return e;
   }
   case Lexer::END:
     return 1;
   default:
     return error("primary expected");
 }
}
```

Объявление ***using*** вводит локальный синоним.

Полезно локальные синонимы делать настолько локальными, насколько это возможно, чтобы избегать конфликта имен. Однако все функции синтаксического анализатора используют одни и те же наборы имен из других модулей. Поэтому мы можем поместить объявления ***using*** непосредственно внутрь определения пространства имен ***Parser***:

```
namespace Parser
{
 double prim(bool);
 double term(bool);
 double expr(bool);
 using Lexer::get_token;         // использовать get_token из Lexer
 using Lexer::curr_tok;          // использовать curr_tok из Lexer
 using Error::error;             // использовать error из Error
}
```

Это позволяет упростить код функций из пространства имен ***Parser*** практически до их первоначального состояния (§6.1.1).

```
double Parser::term(bool get)    // умножение и деление
{
  double left=prim(get);

  for(;;)
    switch(curr_tok)
    {
      case Lexer::MUL:
        left*= prim(true);
        break;
      case Lexer::DIV:
        if(double d = prim(true))
        {
          left /= d;
          break;
        }
        return error("divide by 0");
      default:
        return left;
    }
}
```

Я мог бы с помощью объявлений *using* внести в пространство имен ***Parser*** также и имена лексем из ***Lexer***. Но я оставил их полную запись с явными квалификаторами как напоминание о зависимости между ***Parser*** и ***Lexer***.

8.2.3. Директивы using

А что если бы мы захотели упростить функции из пространства имен ***Parser*** в *точности* до их первоначальных версий? Это было бы вполне разумной задачей для большой программы, которая конвертируется из предыдущей версии с малой модульностью.

Директивы using (*using-directive*) позволяют сделать имена из некоторых пространств имен столь же доступными, как если бы они объявлялись вне этих пространств имен (§8.2.8). Например:

```
namespace Parser
{
  double prim(bool);
  double term(bool);
  double expr(bool);

  using namespace Lexer;     // сделать все имена из Lexer доступными
  using namespace Error;     // сделать все имена из Error доступными
}
```

Это позволяет нам написать функции из пространства имен ***Parser*** точно так же, как мы это делали ранее (§6.1.1):

```
double Parser::term (bool get)    // умножение и деление
{
  double left = prim (get) ;

  for ( ; ; )
    switch (curr_tok)
    {
      case MUL:
        left *= prim (true) ;
        break;
      case DIV:
        if (double d = prim (true) )
        {
          left /= d;
          break;
        }
        return error ("divide by 0") ;
      default:
        return left;
    }
}
```

Директивы ***using***, примененные глобально, являются удобным инструментом для переделки готового кода (§8.2.9), а в иных случаях их лучше избегать. Эти же директивы, примененные внутри определений пространств имен, являются средством их композиции (§8.2.8). В теле функций (и только там) ***директивы using*** можно смело применять с целью улучшения внешнего вида кода (§8.3.3.1).

8.2.4. Множественные интерфейсы

Ясно, что данное выше определение пространства имен ***Parser*** не является интерфейсом, предназначенным для пользователей синтаксического анализатора (***парсера***). Скорее, это набор объявлений, необходимый для удобной разработки функций синтаксического анализатора. А интерфейс для пользователя должен быть существенно более простым:

```
namespace Parser
{
  double expr (bool) ;
}
```

К счастью, можно определить два *пространства имен* ***Parser***, чтобы использовать каждое из них там, где нужно. В совокупности эти пространства имен обеспечивают две вещи:

1. Общую среду для функций парсера
2. Внешний интерфейс для пользователей парсера

Так, управляющая функция ***main***() должна видеть лишь

```
namespace Parser            // интерфейс для пользователей
{
  double expr (bool) ;
}
```

А функции, в совокупности реализующие синтаксический анализатор, должны видеть пространство имен, составляющее интерфейс разработки калькулятора:

```
namespace Parser                // интерфейс для разработчиков
{
  double prim (bool) ;
  double term (bool) ;
  double expr (bool) ;

  using Lexer : :get_token;     // использовать get_token из Lexer
  using Lexer : :curr_tok;      // использовать curr_tok из Lexer
  using Error : :error;         // использовать error из Error
}
```

или в графическом виде:

Здесь стрелки означают отношение «полагается на указанный интерфейс».

Parser' — это «малый» интерфейс для пользователей. Имя ***Parser'*** (читается «Парсер штрих») не является идентификатором C++; оно выбрано намеренно с целью подчеркнуть, что в программе для этого интерфейса отдельного уникального имени нет. Это не приводит к путанице, так как программисты привыкли давать разные имена для принципиально разных интерфейсов, и потому что физическая организация программы (§9.3.2) естественным образом обеспечивает изоляцию имен (в файлах).

Интерфейс для разработчиков шире такового для пользователей. Если бы это был интерфейс разработки реальной программы большого размера, то он еще и менялся бы чаще интерфейса для пользователей. Важно, что пользователи модуля (в нашем случае функция ***main***(), использующая модуль ***Parser***) изолированы от таких изменений.

В принципе, нет необходимости в двух пространствах имен для отражения двух различных интерфейсов, но при желании это можно делать. Вообще, разработка интерфейса является одной из важнейших задач проектирования, в которой можно многое приобрести, и многое потерять. Есть смысл обсудить, чего собственно мы хотим достигнуть, и есть ли альтернативные пути достижения того же самого.

Предложенная нами схема решения является самой простой из всех возможных, а часто и самой лучшей. Ее основным недостатком является совпадение имен двух интерфейсов при отсутствии у компилятора достаточной информации для контроля их согласованности между собой. И даже в этом случае компилятор отслеживает многие ошибки. Более того, заметную долю ошибок, пропущенных компилятором, отлавливает компоновщик.

Предложенное решение используется далее при обсуждении физического разбиения на модули (§9.3), и я его рекомендую, если нет дополнительных логических ограничений (см. также §8.2.7).

8.2.4.1. Альтернативы интерфейсам

Главная цель применения интерфейсов — уменьшение зависимостей между частями программы. Минимально достаточные интерфейсы приводят к программам, которые легко понять, которые хорошо скрывают внутренние данные, которые легко модернизировать, и которые даже компилируются быстрее.

Следует иметь в виду, что в отношении зависимостей и компиляторам, и программистам часто свойственен упрощенный взгляд на проблему: «Если определение видимо в точке X, то любой код в точке X зависит от содержимого этого определения». В общем случае, однако, дела обстоят не столь плохо, поскольку на самом деле большая часть кода не зависит от большей части определений. К примеру, в нашем случае мы имеем следующее:

```
namespace Parser            // интерфейс для реализации
{
  // ...
  double expr (bool) ;
  // ...
}

int main ()
{
  // ...
  Parser::expr (false) ;
  // ...
}
```

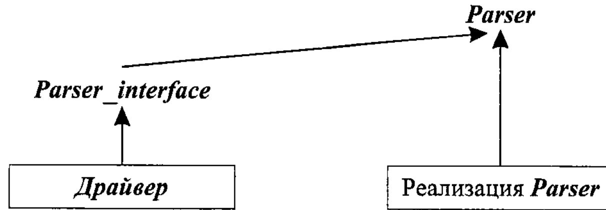
Здесь функция *main()* зависит только от *Parser::expr()*, но чтобы выявить этот факт, нужно время, умственные усилия, практическая проверка гипотезы и т.д. Как следствие, в реальных программах больших размеров и компиляторы, и программисты перестраховываются и считают, что если зависимость может быть, то она действительно имеет место. Надо сказать, этот подход разумный.

Таким образом, желательно организовать программу так, чтобы уменьшить набор потенциальных зависимостей до набора реальных зависимостей.

Сначала пробуем очевидное: определяем пользовательский интерфейс к парсеру в терминах имеющегося интерфейса разработчика:

```
namespace Parser            // интерфейс для реализации
{
  // ...
  double expr (bool) ;
  // ...
}
namespace Parser_interface{ // интерфейс для пользователей
{
  using Parser::expr;
}
```

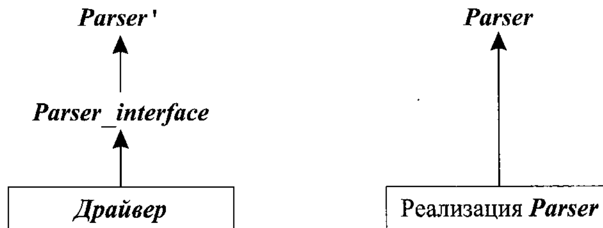
Ясно, что пользователи *Parser_interface* зависят только (причем косвенно) от *Parser::expr()*. Однако на первый (поверхностный) взгляд имеет место следующая схема зависимостей:



Теперь уже кажется, что пользователь (управляющая программа) зависит от любых изменений в ***Parser***, от которых мы его хотели бы изолировать. Даже такая кажущаяся зависимость нежелательна, и мы явным образом ограничиваем ***Parser_interface*** лишь имеющей отношение к пользователям частью интерфейса ***Parser*** (ранее мы это называли ***Parser'***):

```
namespace Parser                // интерфейс для пользователей
{
  double expr (bool) ;
}

namespace Parser_interface      // отдельно поименованный интерфейс для пользователей
{
  using Parser::expr;
}
```

или графически:



Для проверки согласованности ***Parser*** и ***Parser'*** между собой мы снова полагаемся на совокупность всех средств компиляции и компоновки, а не на работу одного лишь компилятора над одной единицей компиляции. Настоящее решение отличается от такового из §8.2.4 лишь наличием дополнительного пространства имен ***Parser_interface***. При желании мы могли бы явным образом объявить функцию ***expr***() непосредственно в ***Parser_interface***:

```
namespace Parser_interface
{
  double expr (bool) ;
}
```

Теперь нет необходимости держать ***Parser*** в области видимости в момент определения ***Parser_interface***. Это нужно лишь в момент определения функции ***Parser_interface::expr***():

```
double Parser_interface::expr(bool get)
{
  return Parser::expr(get);
}
```

Последний вариант можно отобразить графически следующим образом:

Все зависимости теперь сведены к минимуму, все конкретизировано и именовано надлежащим образом. Однако в большинстве реальных практических случаев, с которыми мне приходится сталкиваться, это решение чрезмерно сложно и выходит за пределы необходимого.

8.2.5. Как избежать конфликта имен

Пространства имен предназначены для отражения внутренней структуры кода. Простейшей целью применения пространств имен является разграничение кода, написанного одним программистом, от кода, написанного кем-то еще. Такое разграничение кода имеет огромное практическое значение.

Действительно, на базе одной лишь глобальной области видимости неоправданно сложно составлять программу из отдельных частей. Проблема состоит в том, что предположительно отдельные части могут определять одни и те же имена. При объединении частей в программу происходит конфликт имен. Рассмотрим пример:

```
// my.h:
  char f(char);
  int f(int);
  class String { /* ... */ };

// your.h:
  char f(char);
  double f(double);
  class String { /* ... */ };
```

При наличии таких определений трудно составить единую программу, используя одновременно и ***my.h***, и ***your.h***. Очевидное решение состоит в том, чтобы включить каждый из наборов объявлений в его собственное пространство имен:

```
namespace My
{
  char f(char);
  int f(int);
  class String { /* ... */ };
}
```

```
namespace our
{
  char f(char) ;
  double f(double) ;
  class String {/* ... */} ;
}
```

Теперь можно использовать объявления из ***My*** и ***Your*** либо при помощи явной квалификации (§8.2.1), либо применением объявлений ***using*** (§8.2.2) или директив ***using*** (§8.2.3).

8.2.5.1. Неименованные пространства имен

Часто бывает полезно обернуть набор объявлений объемлющим пространством имен просто ради того, чтобы избежать потенциального конфликта имен. В таком случае главная цель состоит в локализации имен, а не в предоставлении интерфейса к пользовательскому коду. Например:

```
#include "header.h"
namespace Mine
{
  int a;
  void f() {/* ... */}
  int g() {/* ... */}
}
```

Так как мы не планируем делать имя ***Mine*** доступным вне локального контекста, выбор имени становится определенной обузой, к тому же отягощенной возможностью конфликта с другими глобальными именами. В этом случае имя пространства имен можно просто опустить:

```
#include "header.h"
namespace
{
  int a;
  void f() {/* ... */}
  int g() {/* ... */}
}
```

Доступ к членам *неименованного пространства имен* (*unnamed namespace*) осуществляется в локальном контексте без квалификации, так что предыдущее объявление эквивалентно следующему объявлению

```
namespace $$$
{
  int a;
  void f() {/* ... */}
  int g() {/* ... */}
}
using namespace $$$;
```

плюс соответствующая директива ***using***, а $$$ — некоторое уникальное для текущей области видимости имя.

Неименованные пространства имен в разных единицах компиляции разные по определению, так что, как и задумано, невозможно обратиться к члену неименованного пространства имен из другой единицы компиляции.

8.2.6. Поиск имен

Чаще всего функция с аргументом типа *T* определяется в том же пространстве имен, что и тип *T*. Поэтому, если функцию не удается найти в контексте, где она используется, поиск продолжается в пространстве имен ее аргументов. Например:

```
namespace Chrono
{
  class Date{ /* ... */ };
  bool operator==(const Date&, const std::string&);
  std::string format(const Date&);    // выдать строковое представление
  // ...
}

void f(Chrono::Date d, int i)
{
  std::string s = format(d);          // Chrono::format()
  std::string t = format(i);          // error: нет format() в области видимости
}
```

Это правило экономит усилия программиста, так как не требует ввода квалифицированных имен, и в то же время не засоряет область видимости так, как это делает директива *using* (§8.2.3). Рассмотренное правило особо полезно в случае операндов перегружаемых операций (§11.2.4) и аргументов шаблонов (§C.13.8.4), где явная квалификация может быть особо обременительной.

Важно, однако, чтобы при этом само пространство имен находилось в области видимости, а функция была объявлена до ее использования.

Естественно, что функция может иметь аргументы из нескольких пространств имен. Например:

```
void f(Chrono::Date d, std::string s)
{
  if(d == s)
  {
    // ...
  }
  else if(d == "August 4, 1914")
  {
    // ...
  }
}
```

В таком случае, функция ищется в текущей (по отношению к ее вызову) области видимости и в пространствах имен ее аргументов (включая классы аргументов и их базовые классы). После этого ко всем найденным функциям применяются обычные правила разрешения перегрузки (§7.4). Например, для ***d==s*** выполняется поиск ***operator==*** в области видимости, окружающей ***f()***, в пространстве имен ***std*** (где определяется операция == для стандартного библиотечного типа ***string***) и в простран-

стве имен *Chrono*. При этом находится операция *std*::*operator*==(), но у нее нет аргумента типа *Date*, так что используется *Chrono*::*operator*==(), имеющая такой аргумент. См. также §11.2.4.

Когда член класса вызывает именованную функцию, остальные члены этого класса и его базовых классов имеют предпочтение перед функциями, выбираемыми на основании типов аргументов; ситуация же с перегруженными операциями иная (§11.2.1, §11.2.4).

8.2.7. Псевдонимы пространств имен

Если давать пространствам имен короткие имена, то они могут войти в конфликт друг с другом:

```
namespace A     //слишком коротко (может войти в конфликт)
{
  // ...
}

A::String s1 = "Grig";
A::String s2 = "Nielsen";
```

В то же время, длинные имена использовать в реальном коде не очень удобно:

```
namespace American_Telephone_and_Telegraph    // слишком длинное
{
  // ...
}

American_Telephone_and_Telegraph::String s3 = "Grieg";
American_Telephone_and_Telegraph::String s4 = "Nielsen";
```

Дилемма разрешается с помощью коротких *псевдонимов* (*aliases*) для длинных имен:

```
// используем псевдонимы:
namespace ATT = American_Telephone_and_Telegraph;
ATT::String s3 = "Grieg";
ATT::String s4 = "Nielsen";
```

С помощью псевдонимов удобно ссылаться на большой набор объявлений («библиотеку»), точно указывая библиотеку в единственной строчке кода. Например:

```
namespace Lib = Foundation_library_v2r11;
// ...
Lib::set s;
Lib::String s5 = "Sibelius";
```

Это кардинально упрощает проблему смены версии библиотеки. Непосредственно применяя *Lib* вместо *Foundation_library_v2r11*, мы можем легко перейти к версии "*v3r02*" изменением одной лишь инициализации псевдонима *Lib* и перекомпиляцией. Перекомпиляция может выявить возможные рассогласования на уровне исходного кода. В то же время, избыточное применение псевдонимов затрудняет понимание кода и ведет к путанице.

8.2.8. Композиция пространств имен

Часто возникает необходимость скомбинировать интерфейс из ряда существующих интерфейсов. Например:

```
namespace His_string
{
  class String { /* ... */ };
  String operator+ (const String&, const String&) ;
  String operator+ (const String&, const char*) ;
  void fill (char) ;
  // ...
}

namespace Her_vector
{
  template<class T> class Vector { /* ... */ };
  // ...
}

namespace My_lib
{
  using namespace His_string;
  using namespace Her_vector;
  void my_fct (Strin&) ;
}
```

После этого мы можем писать программу в терминах ***My_lib***:

```
void f()
{
  My_lib::String s = "Byron";          // находит My_lib::His_string::String
  // ...
}

using namespace My_lib;

void g (Vector<String>& vs)
{
  // ...
  my_fct (vs[5]) ;
  // ...
}
```

Если явно квалифицированное имя (такое как ***My_lib::String***) не объявляется в указанном пространстве имен, то компилятор ищет его в пространствах имен, указанных через директивы ***using*** (например, в ***His_string***).

Истинное (конечное) пространство имен требуется указывать лишь в определениях:

```
void My_lib::fill (char c)             // error: никакого fill() в My_lib нет
{
  // ...
}
void His_string::fill (char c)         // ok: fill() объявлена в His_string
```

```
{
  // ...
}
void My_lib::my_fct(String& v)      // ok; String есть My_lib::String, в
                                    // смысле His_String::String
{
  // ...
}
```

В идеале, пространство имен должно:

1. Отражать логически связанный набор элементов.
2. Ограничивать доступ пользователей к не предназначенным для них элементам.
3. Не требовать излишних усилий при использовании.

Техника композиции пространств имен, рассматриваемая в текущем и последующих разделах, совместно с препроцессорными директивами *#include* (§9.2.1), обеспечивают необходимую поддержку этих требований.

8.2.8.1. Отбор отдельных элементов из пространства имен

Иногда нам нужен доступ лишь к некоторым элементам из пространства имен. Для этого можно объявить пространство имен, содержащее лишь необходимые элементы. Например, мы могли бы объявить иную версию *His_string*, содержащую только класс *String* и перегруженные операции конкатенации:

```
namespace His_string                // лишь часть из His_string
{
  class String { /* ... */ };
  String operator+(const String&, const String&);
  String operator+(const String&, const char*);
}
```

Однако если я не являюсь разработчиком или программистом, сопровождающим код, то такой подход быстро приведет к хаосу. Изменения в «настоящем» *His_string* никак не отражаются в данном объявлении. Отбор отдельных элементов из некоторого пространства имен следует явным образом выполнять с помощью объявлений *using*:

```
namespace My_string
{
  using His_string::String;
  using His_string::operator+;      // любой + из His_string
}
```

Объявление *using* вносит имя в текущую область видимости. В частности, единственное объявление *using* вносит в область видимости все имеющиеся варианты перегруженных функций.

В таком случае, если разработчик *His_string* внесет новую функцию-член в класс *String*, или добавит новый вариант перегруженной операции конкатенации, то эти изменения станут автоматически доступными пользователям *My_string*. Если же из *His_string* удаляется некоторый элемент или какой-то его элемент изменяется, то ставшее при этом некорректным применение *My_string* сразу же будет обнаружено компилятором (см. также §15.2.2).

8.2.8.2. Композиция пространств имен и отбор элементов

Если объединить композицию пространств имен (при помощи директив ***using***) с отбором элементов (при помощи объявлений ***using***), то будет получена гибкость, достаточная для решения большинства реальных задач. С помощью этих механизмов мы можем обеспечить доступ к определенному набору средств таким образом, чтобы надежно избегать при этом конфликта имен и неоднозначностей. Например:

```
namespace His_lib
{
  class String { /* ... */ };
  template<class T> class Vector{ /* ... */ };
  // ...
}

namespace Her_lib
{
  template<class T> class Vector { /* ... */ };
  class String { /* ... */ };
  // ...
}

namespace My_lib
{
  using namespace His_lib;                    // все из His_lib
  using namespace Her_lib;                    // все из Her_lib

  using His_lib::String;                      // разрешается в пользу His_lib
  using Her_lib::Vector;                      // разрешается в пользу Her_lib
  template<class T> class List { /* ... */ };  // прочее
  // ...
}
```

Имена, объявленные явно в пространстве имен (включая те, что присутствуют в составе объявлений ***using***) имеют приоритет над именами, внесенными с помощью директив ***using*** (см. также §С.10.1). Следовательно, пользователь ***My_lib*** обнаружит, что конфликты имен ***String*** и ***Vector*** разрешаются в пользу ***His_lib::String*** и ***Her_lib::Vector***. Также по умолчанию получает приоритет ***My_lib::List***, независимо от того, объявляется ли в ***His_lib*** или ***Her_lib*** класс ***List*** или нет.

Обычно, я предпочитаю не менять имя при включении его в новое пространство имен. В результате мне не нужно помнить два разных имени для одного и того же элемента. Тем не менее, иногда нужно или удобно ввести новое имя. Например:

```
namespace Lib2
{
  using namespace His_lib;                    // все из His_lib
  using namespace Her_lib;                    // все из Her_lib

  using His_lib::String;                      // разрешается в пользу His_lib
  using Her_lib::Vector;                      // разрешается в пользу Her_lib

  typedef Her_lib::String Her_string;         // переименовываем
```

```
    // "Переименовываем":
    template<class T> class His_vec : public His_lib::Vector<T>{ /* ... */ };
    template<class T> class List { /* ... */ };      // прочее
    // ...
}
```

В языке нет специальных средств для переименования. Приходится использовать стандартные механизмы определения новых сущностей.

8.2.9. Пространства имен и старый код

Миллионы строк кода на C и C++ используют глобальные имена и ранние версии библиотек. Как применить пространства имен, чтобы смягчить проистекающие из-за этого проблемы? Переписать существующий код возможно далеко не всегда, но к счастью, библиотеками языка C можно пользоваться так, как будто они определены в пространствах имен. С библиотеками C++ такой номер не проходит (§9.2.4), но пространства имен разработаны таким образом, что их можно было почти что безболезненно добавлять в существующий код на языке C++.

8.2.9.1. Пространства имен и язык C

Рассмотрим каноническую начальную программу на C:

```
#include <stdio.h>
int main()
{
  printf("Hello, world!\n");
}
```

Вряд ли хорошей идеей будет явная переработка этой программы, так же как и разработка специальных версий стандартных библиотек. Поэтому правила языка для пространств имен разработаны таким образом, что можно относительно легко превратить старый код, не использующий пространств имен, в более структурированную версию, опирающуюся на пространства имен (наша программа калькулятор (§6.1) в целом демонстрирует этот подход).

В качестве одного из способов решения этой задачи, поместим объявления из старых версий файла ***stdio.h*** в пространство имен ***std***:

```
// файл cstdio:
namespace std
{
  int printf(const char* ... );
  //...
}
```

При наличии файла <***cstdio***>, можно обеспечить обратную совместимость, если в новый заголовочный файл <***stdio.h***> поместить следующий код:

```
// файл stdio.h:
#include <cstdio>
using namespace std;
```

Этот вариант файла <***stdio.h***>, опирающийся на директиву ***using***, позволяет успешно скомпилировать старый вариант программы «Hello, world!». К сожалению,

директива ***using*** вносит в глобальную область видимости вообще все имена из пространства имен ***std***. Например:

```
#include <vector>     // избегаем загрязнения глобального пространства
vector v1;            // error: нет "vector" в глобальной области видимости
#include <stdio.h>    // содержит "using namespace std;"
vector v2;            // oops: теперь это работает
```

Поэтому в файле <***stdio.h***> лучше применить объявления ***using***, вносящие в глобальную область видимости лишь элементы, определенные в файле <***cstdio***>:

```
// файл stdio.h:
#include <cstdio>
using std::printf;
// ...
```

Еще одно преимущество состоит в том, что объявление ***using*** предотвращает (случайное или намеренное) определение пользователем нестандартных версий функции ***printf***() в глобальной области видимости. Вообще, я рассматриваю нелокальные директивы ***using*** лишь как переходное средство. В большинстве случаев код в программах, ссылающихся на имена из других пространств имен, лучше выражается при помощи явных квалификаторов и объявлений ***using***.

Связь между пространствами имен и компоновкой рассматривается в §9.2.4.

8.2.9.2. Пространства имен и перегрузка

Перегрузка (§7.4) работает поверх пространств имен. Это существенный момент, помогающий мигрировать от старых версий библиотек к применению пространств имен с минимальными изменениями исходного кода. Например:

```
// старый A.h:
void f(int);
// ...

// старый B.h:
void f(char);
// ...

// старый user.c:
#include "A.h"
#include "B.h"

void g()
{
  f('a');             // вызывается f() из B.h
}
```

Эту программу можно переработать в версию с пространствами имен без изменения содержательной части кода:

```
// новый A.h:
namespace A
{
  void f(int);
  // ...
}
```

```
// новый B.h:
namespace B
{
  void f(char) ;
  // ...
}

// новый user.c:
#include "A.h"
#include "B.h"

using namespace A;
using namespace B;

void g ()
{
  f('a') ;            // вызывается f() из B.h
}
```

Если бы мы хотели оставить файл **user.c** вообще без изменений, то в таком случае можно было бы поместить директивы **using** в заголовочные файлы.

8.2.9.3. Пространства имен открыты

Пространства имен открыты — это означает, что в них можно добавлять имена в пределах нескольких объявлений одного и того же пространства имен:

```
namespace A
{
  int f() ;           // в A присутствует f()
}

namespace A
{
  int g () ;          // в A присутствуют f() и g()
}
```

Отсюда следует, что мы можем реализовывать большие программные фрагменты в рамках одного и того же пространства имен точно так же, как ранее разрабатывались приложения и библиотеки в рамках единственного глобального пространства. Для этого нужно распределить определение пространства имен по нескольким заголовочным файлам и файлам с исходным кодом. Как показано в примере программы калькулятора (§8.2.4), открытость пространств имен позволяет предоставлять разные интерфейсы разным группам пользователей, открывая для них разные части пространства имен. Открытость также помогает при переходе от старого кода к новым его вариантам. Например, следующий код

```
// мой заголовочный файл:
void f() ;            // моя функция
// ...
#include <stdio.h>
int g () ;            // моя функция
// ...
```

может быть переписан без изменения порядка объявлений:

```
// мой заголовочный файл:
namespace Mine
{
  void f() ;              // моя функция
  // ...
}

#include <stdio.h>

namespace Mine
{
  int g() ;               // моя функция
  // ...
}
```

Разрабатывая новые программы, я предпочитаю организовывать множество мелких пространств имен (§8.2.8), нежели помещать большие фрагменты кода в единственное пространство имен. Однако при конвертировании ранних версий больших программ это не практично.

Для определения элемента, ранее объявленного в пространстве имен, лучше использовать синтаксис *Mine*::, чем повторно открывать *Mine*. Например:

```
void Mine::ff()           // error: ff() нет в Mine
{
  // ...
}
```

Компилятор легко обнаруживает продемонстрированную ошибку. Однако ввиду того, что функции могут сразу определяться в объявлении пространства имен, компилятор не сможет обнаружить эквивалентную ошибку в случае повторного открытия Mine:

```
namespace Mine            // повторное открытие Mine с целью определения функции
{
  void ff()               // oops! ff() добавляется в Mine этим определением
  {
    // ...
  }
  // ...
}
```

Действительно, откуда компилятору знать, что на самом деле вы не вводите новую функцию *ff*().

Псевдонимы пространств имен (§8.2.7) могут применяться для квалификации имен элементов пространства при их определении. Однако эти псевдонимы нельзя использовать для повторного открытия пространств имен.

8.3. Исключения

Когда программа составляется из отдельных модулей и, особенно, когда эти модули принадлежат независимо разработанным библиотекам, обработку ошибок следует разделить на две части:

1. На выдачу сообщений об условиях возникновения ошибок, локальная обработка которых невозможна.
2. На собственно обработку ошибок, обнаруженных в любых частях программы.

Автор библиотеки в состоянии обнаруживать ошибки времени выполнения, но, как правило, не имеет понятия о том, что с ними делать. Пользователь библиотеки может знать, как поступать в той или иной ошибочной ситуации, но не может их обнаруживать (иначе это были бы исключительно ошибки в пользовательском коде).

В примере с калькулятором мы избежали этой проблемы, создав программу в виде единого целого. Обработка ошибок встраивалась там как часть единой системы. Но после того, как мы разнесли разные логические части калькулятора по разным пространствам имен, обнаружилось, что все пространства имен зависят от пространства имен ***Error*** (§8.2.2), и что обработка ошибок в ***Error*** полагается на соответствующее поведение остальных модулей в отношении возникающих в них ошибок. Теперь представим, что у нас нет возможности создавать калькулятор в виде единого целого, и что мы хотим избежать тесной связи между ***Error*** и другими модулями. Вместо всего этого предположим, что синтаксический анализатор (парсер) и другие модули разрабатываются в отсутствии информации о том, как управляющая программа будет обрабатывать ошибки.

Несмотря на всю простоту функции ***error***(), определенная стратегия обработки ошибок в нее была заложена:

```
namespace Error
{
  int no_of_errors;

  double error(const char* s)
  {
    std::cerr<< "error: "<< s << '\n';
    no_of_errors++;
    return 1;
  }
}
```

Функция ***error***() выводит сообщение об ошибке, возвращает некоторое предопределенное значение, позволяющее вызвавшему ее коду продолжить работу, а также регистрирует ошибку в переменной ***no_of_errors***. При этом очень важно, что все части программы знают о существовании функции ***error***(), знают как ее вызывать, и что ожидать от этой функции. К программам, составленным из независимо разработанных бибилиотек, невозможно предъявлять столь большие требования.

Исключения являются средством языка C++, позволяющим *отделить информирование об ошибках от их обработки.* В данном разделе исключения описаны очень кратко и только в контексте их использования в программе калькулятора. Более обстоятельное изучение исключений и случаев их применения сосредоточено в главе 14.

8.3.1. Ключевые слова throw и catch

Концепция *исключений* (*exceptions*) введена в язык для генерации сообщений об ошибках. Например:

```
struct Range_error
{
  int i;
  Range_error (int ii) {i=ii; }          // конструктор (§2.5.2, §10.2.3)
};

char to_char (int i)
{
  if (i<numeric_limits<char>: :min ()  | | numeric_limits<char>: :max () <i)   //см §22.2
    throw Range_error () ;

  return c;
}
```

Функция *to_char*() либо осуществляет нормальный возврат значения переменной *i* типа *char*, либо генерирует исключение *Range_error*. Фундаментальная идея заключается в том, что если функция обнаруживает ошибочную ситуацию, с которой не в состоянии справиться сама, то она *генерирует исключение* (*throw exception*) некоторого типа в надежде на то, что вызывающая функция проблему так или иначе уладит. Функция, способная решить такую проблему, явным образом указывает намерение *перехватывать исключение* (*catch exception*) указанного типа. Например, для вызова функции *to_char*() и перехвата генерируемых ею исключений, нужно написать следующий код:

```
void g (int i)
{
  try
  {
    char c = to_char (i) ;
    // ...
  }
  catch (Range_error)
  {
    cerr<< "oops\n";
  }
}
```

Конструкция

```
catch ( /* ... */ )
{
  // ...
}
```

называется *обработчиком исключений* (*exception handler*). Эта конструкция может располагаться только сразу после блока, помеченного ключевым словом *try*, или сразу после другого обработчика исключений; *catch* является ключевым словом языка C++. В круглых скобках располагается объявление того же типа, что и объявления аргументов функций. То есть оно специфицирует тип объектов, которые могут быть перехвачены обработчиком, а также может именовать этот объект (опционально). Например, если бы мы захотели узнать значение сгенерированного в качестве исключения объекта типа *Range_error*, мы бы снабдили этот объект именем так же, как мы снабжаем именами аргументы функций. Например:

```
void h (int i)
{
  try
  {
    char c = to_char (i) ;
    // ...
  }
  catch (Range_error x)
  {
    cerr<< "oops: to_char (" << x.i << ") \n";
  }
}
```

Если код в *try-блоке* (*try-block*), или код, вызываемый из него, генерирует исключение, будут проверяться все обработчики, относящиеся к данному *try*-блоку. Если тип сгенерированного исключения совпадает с объявленным в одном из обработчиков, то выполняется код этого обработчика. Если же ни один из обработчиков не подошел по этому критерию, то все обработчики игнорируются, и в этом случае *try*-блок ведет себя так же, как и обычный блок кода. Если сгенерированное исключение не перехватывается обработчиками, то программа принудительно закрывается (§14.7).

В своей основе, обработка исключений сводится к передаче управления из вызывающей функции в специально указанный (назначенный) код. Если требуется, то дополнительная информация об ошибке может быть передана в вызывающую функцию. Программисты на С могут рассматривать механизм обработки исключений как улучшенную замену для *setjmp*/*longjmp* (§16.1.2). Важная тема взаимоотношений обработки исключений и классов рассматривается в главе 14.

8.3.2. Обработка нескольких исключений

В общем случае в программе могут возникать разные *ошибки на этапе выполнения* (*run-time errors*). Этим ошибкам можно сопоставить несколько типов исключений с различающимися именами. Я предпочитаю для обработки исключений использовать специально предназначенные для этого типы (чтобы предельно ясно выразить через них свои намерения). В частности, с этой целью я никогда не использую встроенные типы (вроде *int*). В большой программе не будет эффективных способов удостовериться, что такие обработчики имеют дело исключительно с предназначенными для них ошибками. То есть возможна путаница в обработке ошибок из разных источников.

Наша программа калькулятор (§6.1) должна обрабатывать два типа ошибок: синтаксические ошибки и попытки деления на нуль. Нет необходимости передавать обработчику ошибки деления на нуль какую-либо дополнительную информацию, так что этот тип ошибок может быть представлен следующим простейшим (пустым) типом:

```
struct Zero_divide { };
```

С другой стороны, обработчик синтаксических ошибок наверняка захочет узнать, каков характер ошибки. В этом случае мы передаем строку:

```
struct Syntax_error
{
  const char* p;
  Syntax_error (const char* q) {p = q; }
};
```

Для удобства я добавил здесь в структуру ***конструктор*** (§2.5.2, §10.2.3).

Пользователь синтаксического анализатора может осуществить обработку этих двух исключений, добавив обработчики обоих типов к *try*-блоку. По ситуации будет выполняться один из них. По выходе из тела обработчика управление передается коду, следующему за самым последним обработчиком в списке:

```
try
{
  //...
  expr (false) ;
  // сюда попадаем только если expr() не возбуждает исключение
  // ...
}
catch (Syntax_error)
{
  // обработка синтаксической ошибки
}
catch (Zero_divide)
{
  // делаем что-то в ответ на попытку деления на нуль
}
// сюда попадаем, если expr не вызвала исключения или если были исключения
// Syntax_error или Zero_divide, а их обработчики не изменили потока управления
// с помощью return, throw или каким-либо иным способом.
```

Список обработчиков выглядит почти как оператор ***switch***, только не требуется оператор ***break***. Это отличие в синтаксисе призвано подчеркнуть тот факт, что каждый обработчик образует отдельную область видимости (§4.9.4).

Функции не обязаны перехватывать все возможные типы исключений. Например, предыдущий *try-блок* не имеет обработчика потенциально возможных исключений от операций ввода. Эти исключения просто оставляются для дальнейшего поиска обработчиков (передаются «наверх» по стеку вызовов функций).

С точки зрения синтаксиса языка исключение считается обработанным сразу же после входа в обработчик. Это сделано для того, чтобы исключения, сгенерированные в теле этого обработчика относились к «вышестоящему» коду, вызвавшему соответствующий *try-блок*. Например, в следующем коде нет бесконечного цикла обработки исключений:

```
class Input_overflow { /* ... */ };

void f()
{
  try
  {
    // ...
  }
```

```
  catch (Input_overflow)
  {
    // ...
    throw Input_overflow () ;
  }
}
```

Обработчики исключений могут быть вложенными. Например:

```
class XXII { /* ... */ } ;

void f()
{
  // ...
  try
  {
    // ...
  }
  catch (XXII)
  {
    try
    {
      // что-нибудь нетривиальное
    }
    catch (XXII)
    {
      // код нетривиального обработчика сам "грохнулся"
    }
  }
  // ...
}
```

Однако такая вложенность редко встречается в коде, написанном человеком, и вообще-то является плохим стилем.

8.3.3. Исключения в программе калькулятора

Отталкиваясь от базового механизма обработки исключений, мы можем переработать код калькулятора из §6.1 таким образом, чтобы отделить обработку ошибок времени выполнения от основной логики программы. Это приведет к такой организации кода, которая хорошо отражает структуру реальных программ, построенных из отдельных, слабо зависимых частей.

Сначала устраним функцию *error*(). Вместо этого функции парсера будут знать лишь типы, сигнализирующие об ошибках:

```
namespace Error
{
  int no_of_errors;

  struct Zero_divide
  {
    Zero_divide() { no_of_errors++; }
  };
```

```
  struct Syntax_error
  {
    const char* p ;
    Syntax_error (const char* q) { p = q; no_of_errors++; }
  };
}
```

Парсер (синтаксический анализатор) генерирует три типа ошибок:

```
Lexer::Token_value Lexer::get_token()
{
  using namespace std;              // для доступа к input, isalpha() и т.д.(§6.1.7)

  // ...
    default:                        // NAME, NAME =, или ошибка
      if(isalpha(ch))
      {
        string_value = ch;
        while(input->get(ch) && isalnum(ch)) string_value.push_back(ch);
        input->putback(ch);
        return curr_tok=NAME;
      }
      throw Error::Syntax_error("bad token");
  }
}

double Parser::prim(bool get)       // первичные выражения
{
  // ...
    case Lexer::LP:
    {
      double e = expr(true);
      if(curr_tok != Lexer::RP) throw Error::Syntax_error(" ')' expected");
      get_token ();                 // пропустить ')'
      return e;
    }
    case Lexer::END:
      return 1;
    default:
      throw Error::Syntax_error("primary expected");
  }
}
```

При обнаружении ошибочной ситуации оператором ***throw*** генерируется исключение с передачей управления на обработчки, определенный где-нибудь в вызывающем (прямо или косвенно) коде. Оператор ***throw*** передает обработчику также и значение. Например, следующий оператор

```
throw Syntax_error("primary expected");
```

передает обработчику объект типа ***Syntax_error***, содержащий указатель на строку ***"primary expected"***.

Выявление деления на нуль не сопровождается передачей сопутствующих данных:

```
double Parser::term(bool get)     // умножение и деление
{
 // ...
    case Lexer::DIV:
      if(double d = prim(true))
      {
        left /= d;
        break;
      }
      throw Error::Zero_divide();
 // ...
}
```

Теперь можно писать управляющий код, который обрабатывает исключения ***Syntax_error*** и ***Zero_divide***:

```
int main(int argc, char* argv[])
{
 // ...
 while(*input)
 {
  try
  {
    Lexer::get_token();
    if(Lexer::curr_tok == Lexer::END) break;
    if(Lexer::curr_tok == Lexer::PRINT) continue;
    cout<< Parser::expr(false) << '\n';
  }
  catch(Error::Zero_divide)
  {
    cerr<< "attempt to divide by zero\n";
    if(Lexer::curr_tok != Lexer::PRINT) skip();
  }
  catch(Error::Syntax_error e)
  {
    cerr<< "syntax error:" << e.p << "\n";
    if(Lexer::curr_tok != Lexer::PRINT) skip();
  }
 }

 if(input != &cin) delete input;
 return no_of_errors;
}
```

Во всех случаях, кроме ошибки в самом конце ограниченного лексемой PRINT выражения (т.е. концом строки или точкой с запятой), функция ***main***() вызывает «восстанавливающую» функцию ***skip***(). Эта функция пытается вернуть парсер в нормальное состояние, пропуская для этого символы до тех пор, пока не встретится конец строки или точка с запятой. Очевидными кандидатами на членство в пространстве имен ***Driver*** являются ***input*** и ***skip***():

```
namespace Driver
{
  std::istream* input;
  void skip();
}

void Driver::skip()
{
  while (*input)        // пропускать символы, пока не встретятся newline или;
  {
    char ch;
    input->get(ch);

    switch (ch)
    {
      case '\n':
      case ';':
        return;
    }
  }
}
```

Код функции *skip*() намеренно написан на более низком уровне абстракции, чем парсер, с целью избежать проблемы генерации новых исключений парсера во время обработки текущих исключений парсера.

Возникает искушение слить воедино пространства имен *Error* и *Driver*, ибо обработка ошибок и операции управляющей программы, такие как ввод/вывод, тесно связаны. Например, подсчет числа ошибок и выдача сообщений о них вполне могли бы быть одной из задач управляющей программы. Однако лучше сохранять явное размежевание между сущностями, разными с логической точки зрения.

Функцию *main*() я не стал помещать в пространство имен *Driver*. Глобальная функция *main*() является стартовой функцией программы (§3.2); нет никакого смысла помещать ее в иное пространство имен. Будь наша программа существенно большего размера, заметная часть кода из функции *main*() была бы перемещена в отдельную функцию из пространства имен *Driver*.

8.3.3.1. Альтернативные стратегии обработки ошибок

Оригинальный код обработки ошибок был короче и элегантнее версии, использующей исключения. Однако это было достигнуто за счет более тесной связи между частями программы. Такой подход плохо масштабируется на большие программы, составленные из независимо разработанных библиотек.

Можно рассмотреть вариант с отказом от отдельной функции обработки ошибок *skip*() за счет введения переменной состояния в функции *main*(). Например:

```
int main(int argc, char* argv[])     // пример плохого стиля
{
  // ...
  bool in_error=false;

  while (*Driver::input)
  {
    try
```

```
    {
      Lexer::get_token();
      if(Lexer::curr_tok == Lexer::END) break;
      if(Lexer::curr_tok == Lexer::PRINT)
      {
        in_error = false;
        continue;
      }
      if(in_error == false) cout<<Parser::expr(false) << '\n';
    }
    catch(Error::Zero_divide)
    {
      cerr<< "attempt to divide by zero\n";
      in_error = true;

    }
    catch(Error::Syntax_error e)
    {
      cerr<< "syntax error:" << e.p<< "\n";
      in_error = true;
    }
  }
  if(Driver::input != &std::cin) delete Driver::input;
  return Error::no_of_errors;
}
```

Я считаю это плохой идеей по нескольким причинам:

1. Переменные состояния служат источником путаницы и ошибок, особенно если их действие распространяется на большие участки кода. В частности, я считаю версию функции ***main***() с переменной состояния ***in_error*** менее читабельной, чем версия с функцией ***skip***().
2. Стратегия раздельного сосуществования «нормального» кода и кода обработки ошибок в общем случае правильная.
3. Реализация кода обработки ошибок на том же уровне абстракции, что и код, порождающий ошибки, опасна; обрабатывающий ошибки код может снова породить те же самые ошибки, что вызвали текущую их обработку. Я предлагаю в качестве упражнения выяснить, как это может произойти в версии функции ***main***() с ***in_error*** (§8.5[7]).
4. Работы требуется больше при добавлении к «нормальному» коду кода обработки ошибок, чем при добавлении отдельных процедур обработки ошибок.

Обработка исключений предназначена для решения нелокальных по своей природе проблем. Если ошибку можно обработать локально, так и следует поступать практически всегда. Например, нет никаких причин использовать исключения для обработки ошибки, связанной со слишком большим количеством аргументов командной строки:

```
int main (int argc, char* argv[])
{
 using namespace std;
 using namespace Driver;

 switch (argc)
 {
   case 1:                                // чтение из стандартного потока
     input = &cin;
     break;
   case 2:                                // чтение командной строки
     input= new istringstream (argv[1]) ;
     break;
   default:
     cerr<< "too many arguments\n";
     return 1;
 }
 // то же, что и раньше
}
```

Далее исключения обсуждаются в главе 14.

8.4. Советы

1. Пользуйтесь пространствами имен для отражения логической структуры; §8.2.
2. Помещайте каждое нелокальное имя, кроме ***main*** (), в некоторое пространство имен; §8.2.
3. Проектируйте пространства имен таким образом, чтобы пользоваться ими было удобно и чтобы не было случайного доступа к пространствам имен, не имеющим отношения к вашей задаче; §8.2.4.
4. Избегайте слишком коротких названий для пространств имен; §8.2.7.
5. При необходимости применяйте псевдонимы для слишком длинных названий пространств имен; §8.2.7.
6. Не нагружайте пользователей ваших пространств имен слишком сложными обозначениями; §8.2.2, §8.2.3.
7. Применяйте квалифицированные имена вида ***Namespace::member*** в определениях членов пространства имен; §8.2.8.
8. Используйте директиву ***using namespace*** только в переходных целях или в локальной области видимости; §8.2.9.
9. Используйте исключения для разъединения кода обработки ошибок и нормального кода; §8.3.3.
10. Для исключений применяйте пользовательские, а не встроенные типы; §8.3.2.
11. Не пользуйтесь исключениями, когда достаточно локального контролирующего кода; §8.3.3.1.

8.5. Упражнения

1. (*2.5) Напишите модуль, реализующий двусвязный список элементов типа ***string*** в стиле модуля ***Stack*** из §2.4. Проверьте его на списке названий языков программирования. Реализуйте функцию ***sort***() для этого списка, и функцию, изменяющую порядок элементов списка на обратный.
2. (*2) Возьмите не слишком большую программу, которая использует по крайней мере одну библиотеку, не применяющую пространств имен. Модифицируйте программу, чтобы в ней использовалось пространство имен для этой библиотеки. Подсказка: §8.2.9.
3. (*2) Реализуйте программу калькулятор в виде модуля в стиле §2.4, используя пространства имен. Не применяйте директивы ***using***. Ведите учет допущенных вами ошибок. Предложите способ, как можно избежать таких ошибок.
4. (*1) Напишите программу, которая генерирует исключение в одной функции, а перехватывает его в другой.
5. (*2) Напишите программу с функциями, вызывающими друг друга с глубиной вложенности вызовов, равной 10. Снабдите каждую функцию аргументом, сообщающим уровень вложенности, на котором произошла генерация исключения. Заставьте функцию ***main***() перехватывать исключения и выводить информацию о перехваченном исключении. Не забудьте случай, когда исключение перехватывается в той же самой функции, где оно генерируется.
6. (*2) Перепишите программу из §8.5[5] так, чтобы она определяла степень затрат на перехват исключений, сгенерированных на разных уровнях вложенности вызовов функций. Добавьте строковый объект в каждую функцию и замерьте все снова.
7. (*1) Найдите ошибку в первой версии функции ***main***() из §8.3.3.1.
8. (*2) Напишите функцию, которая в зависимости от аргумента либо возвращает некоторое значение, либо генерирует исключение в виде объекта, имеющего это же значение. Замерьте разницу во времени исполнения этих двух вариантов.
9. (*2) Модифицируйте версию калькулятора из §8.5[3] так, чтобы использовались исключения. Фиксируйте свои ошибки. Предложите способ, как можно избежать таких ошибок.
10. (*2.5) Напишите функции ***plus***(), ***minus***(), ***multiply***() и ***divide***(), контролирующие переполнения (и потерю точности) и генерирующие исключения в этих случаях.
11. (*2) Модифицируйте программу калькулятора так, чтобы она использовала функции из §8.5[10].

9

Исходные файлы и программы

Форма должна следовать за функцией.
— Ле Корбузьер

Раздельная компиляция — компоновка — заголовочные файлы — заголовочные файлы стандартной библиотеки — правило одного определения — компоновка с кодом, написанным не на C++ — компоновка и указатели на функции — использование заголовочных файлов для выражения модульности — единственный заголовочный файл — несколько заголовочных файлов — защита от повторных включений — программы — советы — упражнения.

9.1. Раздельная компиляция

Файл служит традиционной единицей хранения информации на компьютере (в файловой системе) и традиционной единицей компиляции. Хотя в принципе и существуют системы, не хранящие информацию в файлах, и в которых программист не может реализовать программу (и компилировать ее) в виде набора исходных файлов, мы все же ограничимся лишь традиционными системами, ориентированными на файлы.

Обычно невозможно расположить текст всей программы в единственном файле. В частности, большинство стандартных библиотек и операционных систем не поставляется в исходных кодах, так что их код нельзя внедрить в текст пользовательской программы. Даже оставаясь в рамках пользовательских программ (реальных размеров), и неудобно, и непрактично сосредотачивать весь код в единственном файле. Рассредоточение разных частей программы по разным файлам подчеркивает ее логическую структуру, помогает человеку понять программу, а также обеспечивает возможность компилятору отслеживать указанную структуру. Когда единицей компиляции является файл, малейшее его изменение или изменения в файлах, от которых он зависит, требует его перекомпиляции. Даже для программ скромных размеров суммарное время компиляции (и последующих перекомпиляций) заметно уменьшается при разбиении программы на несколько исходных файлов подходящих размеров.

Программист предоставляет в распоряжение компилятора *исходный файл* (*source file*). Затем этот файл препроцессируется, то есть выполняются макроподстановки (§7.8) и директивами **#include** в текст вносится содержимое заголовочных файлов (§2.4.1, §9.2.1). В результате препроцессирования образуется то, что принято называть *единицей трансляции* (*translation unit*). Содержимое единицы трансляции описывается правилами языка C++ и над ним, собственно, и работает компилятор. В настоящей книге я подчеркиваю разницу между исходным файлом и единицей трансляции только тогда, когда нужно различать, что видит программист, а что «видит» компилятор.

Чтобы раздельная компиляция могла быть реализована, программист должен предоставить объявления с информацией о типах, достаточные для того, чтобы можно было выполнить анализ единицы трансляции отдельно от остальных частей программы. Все объявления в программе, состоящей из нескольких раздельно компилируемых частей, должны согласовываться между собой точно так же, как и в программе с единственным исходным файлом. У вас должны быть программные средства, позволяющие убедиться в этом. В частности, компоновщик может обнаружить большую часть рассогласований. *Компоновщик* (*linker*) — это программа, которая связывает воедино раздельно откомпилированные части. Иногда компоновщик называют (неправильно) *загрузчиком* (*loader*). Компоновка может быть полностью завершена до загрузки программы в память компьютера для ее выполнения. С другой стороны, к программе можно добавить новый код («динамическая компоновка») уже после загрузки программы.

Организацию программы в виде нескольких исходных файлов часто называют *физической структурой* (*physical structure*) программы. Физическое разделение программы на отдельные файлы желательно выполнять в соответствии с логической структурой программы. Те же самые соображения о зависимостях, которые помогают осуществлять композицию программ из пространств имен, помогают и при разбиении программ на исходные файлы. В то же время, логическая и физическая структуры программы не обязаны быть идентичными. К примеру, может оказаться удобным разместить определения функций из пространства имен по нескольким исходным файлам, или сосредоточить в единственном файле несколько определений разных пространств имен, или разбросать определение единственного пространства имен по нескольким файлам (§8.2.4).

Далее, мы сначала рассмотрим ряд технических вопросов, связанных с компоновкой, а потом обсудим пару способов разбиения нашей программы-калькулятора (§6.1, §8.2) на отдельные файлы.

9.2. Компоновка (linkage)

Имена функций, классов, шаблонов, пространств имен и перечислений должны согласовываться для всех файлов программы, если только эти имена не объявлены явным образом как локальные.

Программист должен обеспечить правильное объявление имен классов, функций, пространств имен и т.д. в каждой единице трансляции, где они используются, и следить за тем, чтобы все объявления одних и тех же программных объектов были согласованы. Например, рассмотрим два файла:

```
// файл file1.c:
int x = 1;
int f() { /* do something */ }

// файл file2.c:
extern int x;
int f();
void g() {x = f(); }
```

Переменная x и функция *f*(), используемые в функции *g*() из файла ***file2.c***, определены в файле ***file1.c***. Ключевое слово ***extern*** означает, что объявления ***x*** и ***f*()** есть исключительно (и только) объявления, но не определения (§4.9). Если бы в объявлении ***x*** присутствовало инициализирующее выражение, то тогда ***extern*** было бы проигнорировано, поскольку объявление с инициализирующим выражением всегда является определением. Любой объект должен определяться в программе ровно один раз. Объявляться же он может многократно, но типы при этом должны совпадать. Например:

```
// файл file1.c:
int x = 1;
int b = 1;
extern int c;

// файл file2.c:
int x;                  // означает int x = 0;
extern double b;
extern int c;
```

В этом примере имеются три ошибки: переменная x определена дважды, в разных объявлениях переменной ***b*** указаны разные типы, а переменная ***c*** объявлена дважды, но ни разу не определена. Ошибки такого рода (ошибки этапа компоновки) не могут быть обнаружены на этапе компиляции, ибо компилятор работает с каждым файлом раздельно. Большинство таких ошибок позже выявляются компоновщиком. Следует отметить, что переменная, определяемая без инициализатора в глобальной области видимости или в рамках некоторого пространства имен, инициализируется по умолчанию. Это не относится к локальным переменным (§4.9.5, §10.4.2) или к объектам, создаваемым в свободной памяти (в куче — §6.2.6). Например, следующий программный фрагмент содержит две ошибки:

```
// файл file1.c:
int x;
int f() {return x; }

// файл file2.c:
int x;
int g() {return f(); }
```

Вызов функции *f*() в файле ***file2.c*** является ошибкой, так как функция *f*() в этом файле не объявлена. Программа не будет скомпонована также еще и потому, что переменная ***x*** определена дважды. А вот в языке C вызов *f*() в файле ***file2.c*** ошибкой не является (§B.2.2).

Говорят, что имена, которые можно использовать в единицах трансляции, отличных от той, где они определены, имеют *внешнюю компоновку* (*external linkage*).

Все имена в предыдущих примерах компоновались внешним образом. Если же на имя можно ссылаться лишь в единице трансляции, где оно определено, то говорят, что имя имеет *внутреннюю компоновку* (*internal linkage*).

Функции с модификатором ***inline*** (§7.1.1, §10.2.9) должны определяться — посредством идентичных определений (§9.2.3) — в каждой единице трансляции, где они используются (вызываются). Следовательно, следующий пример не просто демонстрирует плохой стиль программирования, он вообще недопустим:

```
// файл file1.c:
inline int f(int i) {return i; }

// файл file2.c:
inline int f(int i) {return i+1; }
```

К сожалению, такого рода ошибки с трудом обрабатываются конкретными реализациями, так что следующий пример (вполне логически корректный) — комбинация внешней компоновки и встраивания, запрещается в угоду компиляторам:

```
// файл file1.c:
extern inline int g (int i) ;
int h (int i) {return g (i) ; }      // error: g() не определена в данной единице трансляции

// файл file2.c:
extern inline int g (int i) {return i+1; }
```

По умолчанию ***const*** (§5.4) и ***typedef*** (§4.9.7) подразумевают внутреннюю компоновку. В результате следующий пример является допустимым (но сбивающим с толку):

```
// файл file1.c:
typedef int T;
const int x = 7;

// файл file2.c:
typedef void T;
const int x = 8;
```

Лучше избегать определения глобальных переменных с внутренней компоновкой. Для обеспечения согласованности лучше помещать глобальные объекты с модификаторами ***const*** или ***inline*** только в заголовочные файлы (§9.2.1).

С помощью ключевого слова *extern* объектам с модификатором ***const*** можно навязать внешнюю компоновку:

```
// файл file1.c:
extern const int a = 77;

// файл file2.c:
extern const int a;
void g ()
{
  cout<< a << '\n' ;
}
```

Функция ***g***() выведет 77.

С помощью анонимных (неименованных) пространств имен (§8.2.5) можно превращать имена в локальные по отношению к единице компиляции. Эффект от неименованных пространств имен очень близок к внутренней компоновке. Например:

```
// файл file1.c:
namespace
{
  class X { /* ... */ };
  void f() ;
  int i;
  // ...
}

// файл file2.c:
class X { /* ... */ };
void f() ;
int i;
// ...
```

Здесь функция *f*() из *file1.c* не та же самая, что *f*() из *file2.c*. Использовать одинаковое имя, являющееся локальным для некоторой единицы трансляции, и обозначающее сущности с внешней компоновкой в других единицах трансляции — значит нарываться на неприятности.

В языке С и в более ранних версиях С++ ключевое слово ***static*** означает (и это может привести к путанице) «использовать внутреннюю компоновку» (§В.2.3). Не применяйте ***static*** иначе как внутри функций (§7.1.2) или классов (§10.2.4).

9.2.1. Заголовочные файлы

Типы во всех объявлениях одной и той же сущности должны быть согласованы. Это означает, что исходный код, подаваемый на вход компилятору и позднее собираемый в единое целое компоновщиком, должен быть согласован. Простой (хотя и несовершенный) метод достижения согласованности объявлений в разных единицах трансляции состоит в применении директив ***#include*** *header files* для внесения интерфейсной информации в исходные файлы с исполняемым кодом и/или определением данных.

Механизм директив ***#include*** является простым текстовым средством, позволяющим собрать воедино разные фрагменты исходного кода в единицу (файл) компиляции. Директива

```
#include "to_be_included"
```

замещает строку, содержащую ***#include***, на содержимое файла ***to_be_included***. Файл должен содержать код на С++, так как он поступит на обработку компилятором языка С++.

Имена стандартных библиотечных заголовочных файлов заключайте в угловые скобки < и > (а не в кавычки). Например:

```
#include <iostream>        // из стандартного каталога заголовочных файлов
#include "myheader.h"      // из текущего каталога
```

Обратите внимание на то, что пробелы внутри угловых скобок или кавычек существенны для выполнимости директивы *#include*:

#include < iostream > // *не будет найден файл <iostream>*

Необходимость компилировать файл каждый раз при его включении куда-либо может показаться несколько экстравагантной, но заголовочные файлы в типичном случае содержат только объявления, а не код, требующий серьезного анализа со стороны компилятора. Более того, все современные реализации обеспечивают некоторую форму *предкомпиляции* (*precompiling*) заголовочных файлов для минимизации усилий, требуемых для их повторных компиляций.

Согласно эмпирическому правилу заголовочные файлы могут содержать следующие элементы:

Именованные пространства имен	namespace N{ /* ... */ }
Определения типов	struct Point {int x, y; };
Объявления шаблонов	template<class T> class Z;
Определения шаблонов	template<class T> class V{ /* ... */ }
Объявления функций	extern int strlen (const char*) ;
Определения встроенных функций	inline char get (char* p) { return *p++; }
Объявления данных	extern int a;
Определения констант	const float pi = 3.141593;
Перечисления	enum Light { red, yellow, green };
Объявления имен	class Matrix;
Директивы включения	# include <algorithm>
Макроопределения	# defineVERSION 12
Директивы условной компиляции	# ifdef__cplusplus
Комментарии	/* проверка на конец файла */

Данное эмпирическое правило не является требованием языка. Оно просто формулирует разумные способы использования заголовочных файлов (директив *#include*) для выражения физической структуры программы. С другой стороны, заголовочные фалы не должны содержать следующего:

Определения обычных функций	char get (char* p) { return *p++; }
Определения данных	int a;
Определения агрегатов	short tbl[ ] = {1,2,3};
Неименованные пространства имен	namespace { /* ... */ }
Экспортируемые определения шаблонов	export template<class T> f(T t) { /* ... */ }

Обычно, заголовочные файлы имеют расширение *.h*, а файлы, содержащие определения функций и/или данных, имеют расширение *.c*. Поэтому на них часто ссылаются как на "*.h* файлы" или "*.c* файлы", соответственно. Также в ходу расширения *.C*, *.cxx*, *.cpp* и *.cc*. В руководстве к вашему компилятору об этом должно быть сказано со всей определенностью.

Причина, по которой в заголовочные файлы рекомендуется включать определения простых констант, а определения агрегатов включать не рекомендуется, заключается в том, что реализациям трудно избежать репликации агрегатов в не-

скольких единицах трансляции. К тому же, более простые конструкции встречаются чаще, а потому для них важнее генерировать более качественный машинный код.

Будет разумным не переусердствовать в изобретении слишком «умных» случаев использования директив *#include*. Мои рекомендации сводятся к тому, чтобы включать лишь полные объявления и определения в глобальной области видимости, в блоках спецификации компоновки и в пространствах имен для конвертирования старого кода (§9.2.2). Как всегда, стоит избегать магических фокусов с макроопределениями. Я, например, страшно не люблю отслеживать ошибки неожиданных результатов макроподстановок из косвенно включаемых заголовочных файлов, о которых я даже, ровным счетом, ничего и не слышал.

9.2.2. Заголовочные файлы стандартной библиотеки

Средства стандартной библиотеки описаны в наборе стандартных заголовочных файлов (§16.1.2). В именах этих заголовочных файлов расширения не используются; заголовочные же они потому, что включаются в текст программы с помощью *#include* <...> (стандартные заголовки требуют угловых скобок, а не кавычек). Отсутствие суффикса *.h* не отражает способа хранения содержимого этих файлов; например, содержимое <*map*> может располагаться в файле *map.h* из стандартного каталога. Более того, вообще не требуется, чтобы стандартные заголовки хранились традиционным образом. Конкретные реализации могут по максимуму использовать свои знания о стандартной библиотеке и ее заголовках, реализуя оптимизированные способы их хранения и представления. Например, компилятор, отталкиваясь от своих знаний о встроенном характере стандартной математической библиотеки (§22.3), может трактовать директиву *#include* <*cmath*> просто как некий логический переключатель (тумблер), открывающий возможность применения в программе стандартных математических функций, а вовсе не как требование чтения конкретного файла.

Для каждого стандартного заголовка <*X.h*> языка С имеется соответствующий стандартный заголовок <*cX*> языка С++. Например, *#include* <*cstdio*> обеспечивает то же, что и *#include* <*stdio.h*>. В типичном случае, содержимое *stdio.h* выглядит следующим образом:

```
#Ifdef __cplusplus       // только для С++ компиляторов (§9.2.4)
namespace std {          // стандарт. биб-ка определена в прост. имен std (§8.2.9)

extern "C" {             // функции stdio имеют компоновку языка С (§9.2.4)
#endif

/* ... */
int printf(const char* ...);
/* ... */

#Ifdef __cplusplus
}
}
// ...
using std::printf;       // делает printf доступным в глобальном пространстве имен
// ...
#endif
```

Таким образом, реальное содержимое (объявления) для обоих языков одинаковое, а пространства имен и режимы компоновки введены так, что этот файл можно использовать из программ на C и C++.

9.2.3. Правило одного определения

Каждый класс, перечисление, шаблон и т.д. должны быть определены в программе ровно один раз.

С практической точки зрения это означает, что должно существовать единственное определение, например, класса, находящееся в каком-либо одном файле. К сожалению, языковые правила не столь просты. Например, определение класса можно сформировать через макроопределение (б-р-р!), или определение класса может быть текстуально включено с помощью директив *#include* сразу в несколько исходных файлов (§9.2.1). Еще хуже то, что файл вообще не является концепцией языков C/C++; могут существовать реализации, не использующие файлы для хранения программ.

В результате правило стандарта, гласящее о том, что определения классов, шаблонов и т.д. должны быть уникальными, на самом деле формулируется сложнее и тоньше. Это правило называют *«правилом одного определения»* (*«the one-definition rule»* — сокращенно *ODR*): два определения класса, шаблона или встраиваемой функции принимаются за экземпляры одного и того же уникального определения тогда и только тогда, если:

1. Они находятся в разных единицах трансляции.
2. Они полексемно идентичны.
3. Смысл этих лексем одинаковый в обеих единицах трансляции.

Например:

```
// файл file1.c:
struct S {int a; char b;};
void f(S*);

// файл file2.c:
struct S {int a; char b;};
void f(S* p) { /* ... */ }
```

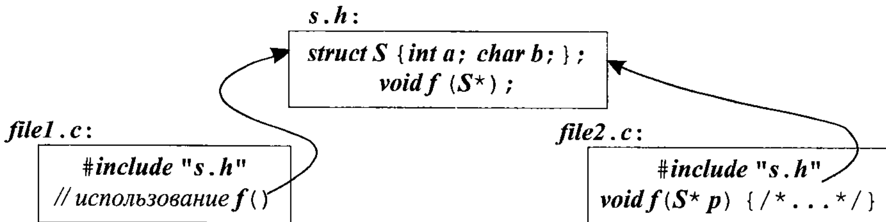
Правило ODR говорит, что данный пример корректен, и что определения *S* в обоих исходных файлах ссылаются на один и тот же класс (структуру). Конечно же, крайне неразумно дважды записывать одно и то же определение таким образом. Кто-нибудь, модифицируя файл *file2.c*, может предположить, что определение *S* из этого файла единственное в программе, и спокойно изменить его. Это приведет к трудноуловимой ошибке.

Правило ODR нацелено на то, чтобы можно было включать определение класса из единственного файлового источника в разные единицы трансляции. Например:

```
// файл s.h:
struct S {int a; char b;};
void f(S*);

// файл file1.c:
#include "s.h"
// используем f() здесь
```

```
// файл file2.c:
#include "s.h"
void f(S* p) {/* ... */}
```

или в графической форме:



Приведем три ошибочных примера, в которых правило ODR нарушается:

```
// файл file1.c:
struct S1 {int a; char b;};
struct S1 {int a; char b;};        // error: повторное определение
```

Здесь ошибка заключается в том, что нельзя дважды определять структуру в одной и той же единице трансляции.

А в следующем примере ошибка состоит в том, что определения структуры ***S2*** в разных файлах содержат разные имена членов структуры.

```
// файл file1.c:
struct S2 {int a; char b;};

// файл file2.c:
struct S2 {int a; char bb;};       // error
```

В третьем ошибочном примере два определения структуры ***S3*** полексемно идентичны, но смысл лексемы ***X*** в разных файлах разный:

```
// файл file1.c:
typedef int X;
struct S3 {X a; char b;};

// файл file2.c:
typedef char X;
struct S3 {X a; char b;};          // error
```

Проверка согласованности определений классов в разных единицах трансляции выходит за пределы возможностей большинства реализаций С++. Как следствие, нарушение правила ODR влечет за собой возникновение тонких ошибок. К сожалению, размещение общих определений в заголовочных файлов с последующих их включением директивами ***#include*** не гарантирует отсутствия последней из представленных в примерах форм нарушения правила ODR. Локальные операторы ***typedef*** и макросы могут менять смысл объявлений из заголовочных файлов:

```
// файл s.h:
struct S {Point a; char b;};

// файл file1.c:
#define Point int
```

```
#include "s.h"
//...

// файл file2.c:
class Point{ /* ... */ };
#include "s.h"
//...
```

Наилучшей защитой от этой проблемы является создание как можно более самодостаточных заголовочных файлов. Например, если бы класс *Point* был объявлен в *s.h*, ошибка была бы обнаружена.

Определение шаблонов можно включать в разные единицы трансляции до тех пор, пока выдерживается правило ODR. Кроме того, если шаблон является экспортируемым (определен в некотором файле с модификатором export), то в ином файле его можно использовать при наличии одного лишь объявления:

```
// файл file1.c:
export template<class T> T twice(T t) {return t+t; }

// файл file2.c:
template<class T> T twice(T t) ; // объявление
int g(int i) {return twice(i) ; }
```

Ключевое слово *export* означает «доступно из другой единицы трансляции» (§13.7).

9.2.4. Компоновка с кодом, написанном не на языке C++

Нередко программы, написанные в основном на C++, содержат фрагменты кода на других языках. И наоборот, программы на других языках используют фрагменты на C++. Добиться корректного взаимодействия таких фрагментов между собой непросто, даже для фрагментов на C++, компилируемых разными компиляторами. Действительно, разные языки и разные компиляторы для одного языка могут по-разному использовать регистры процессора для хранения аргументов, по-разному выполнять раскладку стековых фреймов, могут разниться размещением в памяти отдельных символов и строк символов, целых чисел, чисел с плавающей запятой, могут по-разному передавать имена компоновщику, могут различаться глубиной проверки типов, необходимой компоновщику. Определенную помощь оказывает явное указание *соглашений компоновки* (*linkage convention*) в объявлениях с модификатором *extern*. В следующем примере объявляется функция *strcpy*() стандартной библиотеки языков C/C++ и явно указывается, что компоновку нужно осуществлять по правилам языка C:

```
extern "C" char* strcpy(char*, const char*) ;
```

Эффект от такого объявления отличается от эффекта обычного объявления

```
extern char* strcpy(char*, const chaк*) ;
```

только соглашением о вызове функции *strcpy*().

Директива *extern* "*C*" особо полезна из-за тесной связи между языками C и C++. Важно понимать, что *C* в *extern* "*C*" означает именно соглашение о компоновке, а не язык сам по себе. Часто, extern "*C*" применяется с целью компоновки с процедурами, написанными на языке Fortran или ассемблере, которые удовлетворяют соглашениям языка C.

Директива ***extern "C"*** специфицирует соглашение о компоновке (и только) и не влияет на семантику вызовов функции. В частности, функция, объявленная с ***extern "C"***, по-прежнему подчиняется проверке типов и правилам преобразования аргументов, принятых в C++, а не более слабым правилам языка С. Например:

```
extern "C" int f() ;

int g()
{
  return f(1) ;                    // error: неожиданный аргумент
}
```

Так как добавление ***extern "C"*** к большой группе объявлений утомительно, то предусмотрен механизм группового использования этой директивы:

```
extern "C"
{
  char* strcpy(char*, const char*) ;
  int strcmp(const char*, const char*) ;
  int strlen(const char*) ;
  // ...
}
```

Эту конструкцию часто называют *блоком спецификации компоновки* (*linkage block*) и в нее можно заключить все содержимое заголовочного файла языка С, чтобы сделать его применимым в программах на языке C++. Например:

```
extern "C" {
#include <string.h>
}
```

Такая техника, превращающая заголовочный файл языка С в заголовочный файл языка C++, на практике используется очень часто. Кроме того, применив еще и директивы условной компиляции (§7.8.1), можно получить заголовочный файл, одинаково пригодный для С и C++:

```
#Ifdef __cplusplus
extern "C" {
#endif

char* strcpy(char*, const char*) ;
int strcmp(const char*, const char*) ;
int strlen(const char*) ;
// ...

#Ifdef __cplusplus
}
#endif
```

Предопределенный макрос ***__cplusplus*** позволяет исключить конструкции C++, когда файл используется в качестве заголовочного файла С.

В блоке спецификации компоновки допускаются любые объявления:

```
extern "C" {
int g1;                    // определение
extern int g2;             // объявление (но не определение)
}
```

Никакого дополнительного влияния на области видимости и время жизни переменных при этом не оказывается, так что ***g1*** по-прежнему является глобальной переменной, причем эта переменная определена, а не просто объявлена. Чтобы объявить переменную (а не определить ее), нужно явным образом использовать модификатор ***extern***. Например:

```
extern "C" int g3;         // объявление (не определение)
```

Выглядит немного странно, но на самом деле, это просто следствие неизменности смысла объявлений с модификатором extern при добавлении "*C*" (аналогично неизменности смысла содержимого файла при заключении его в блок спецификации компоновки).

Имя, которое должно компоноваться в стиле языка С, можно объявлять в пространстве имен. Это влияет лишь на то, как имя видимо в коде С++, но не на его представление компоновщику. Функция ***printf***() из пространства имен ***std*** служит типичным примером:

```
#include <cstdio>

void f()
{
  std::printf("Hello, ");   // ok
  printf("world!\n");       // error: нет глобальной printf()
}
```

Несмотря на обращение в виде ***std::printf***(), вызовется-то по-прежнему старая добрая стандартная функция ***printf***() языка С (§21.8).

Это позволяет нам заключать библиотеки с С-компоновкой в произвольные пространства имен и не засорять единственное глобальное пространство имен. Подобная гибкость отсутствует для библиотек, содержащих глобальные определения функций с компоновкой в стиле С++. Причина состоит в том, что компоновка С++-сущностей учитывает пространства имен, и объектные файлы в явном виде содержат информацию об этом (то есть используются пространства имен или нет).

9.2.5. Компоновка и указатели на функции

При смешении фрагментов кода на С и С++ часто приходится передавать указатели на функции одного языка функциям другого языка. Если обе реализации двух языков имеют общие соглашения о компоновке и вызове функций, то проблем при этом не возникает. В общем же случае на это рассчитывать нельзя, так что требуется предпринимать специальные меры, гарантирующие, что вызов функции будет осуществляться надлежащим образом.

Когда соглашение о компоновке указывается в объявлении, то его действие распространяется на все типы, имена и переменные в этом объявлении. Это делает

возможными странные на первый взгляд, но иногда важные комбинации соглашений о компоновке. Например:

```
typedef int (*FT) (const void*, const void*);                    // FT имеет C++ компоновку

extern "C" {
  typedef int (*CFT) (const void*, const void*);                 // CFT имеет C компоновку
  void qsort(void* p, size_t n, size_t sz, CFT cmp);             // cmp имеет C компоновку
}

void isort(void* p, size_t n, size_t sz, FT cmp);                // cmp имеет C++ компоновку
void xsort(void* p, size_t n, size_t sz, CFT cmp);               // cmp имеет C компоновку
extern "C" void ysort(void* p, size_t n, size_t sz, FT cmp);     // cmp имеет C++ ком-ку

int compare(const void*, const void*);                           // compare() имеет C++ ком-ку
extern "C" int ccmp(const void*, const void*);                   // ccmp() имеет C компоновку

void f(char* v, int sz)
{
  qsort(v, sz, 1, &compare);      // error
  qsort(v, sz, 1, &ccmp);         // ok

  isort(v, sz, 1, &compare);      // ok
  isort(v, sz, 1, &ccmp);         // error
}
```

Реализации C и C++ с одинаковыми соглашениями о вызовах могут принять случаи, отмеченные комментарием *//error*, как языковые расширения.

9.3. Применяем заголовочные файлы

Для иллюстрации практического применения заголовочных файлов я сейчас рассмотрю ряд альтернативных способов выражения физической структуры нашей программы калькулятора (§6.1, §8.2).

9.3.1. Единственный заголовочный файл

Простейшим решением проблемы разбиения программы на несколько файлов является размещение определений в подходящем числе .c файлов и помещение всех нужных для их взаимодействия объявлений типов в единственный заголовочный файл, подлежащий включению во все остальные файлы директивой ***#include***. Для программы калькулятора можно использовать пять ***.c*** файлов — ***lexer.c***, ***parser.c***, ***table.c***, ***error.c*** и ***main.c*** — для хранения функций и определений данных, и один заголовочный файл ***dc.h*** для хранения объявлений имен, используемых более чем в одном ***.c*** файле.

Вот возможное содержимое файла ***dc.h***:

```
// файл dc.h:

namespace Error
{
  struct Zero_divide { };
  struct Syntax_error
```

```
  {
    const char* p;
    Syntax_error(const char* q) {p = q; }
  };
}

#include <string>

namespace Lexer
{
  enum Token_value
  {
    NAME,        NUMBER,        END,
    PLUS='+',    MINUS='-',     MUL='*',     DIV='/',
    PRINT=';',   ASSIGN='=',    LP='(',      RP=')'
  };

  extern Token_value curr_tok;
  extern double number_value;
  extern std::string string_value;
  Token_value get_token();
}

namespace Parser
{
  double prim(bool get);      // обработка первичных выражений
  double term(bool get);      // умножение и деление
  double expr(bool get);      // сложение и вычитание

  using Lexer::get_token;
  using Lexer::curr_tok;
}

#include <map>

extern std::map<std::string, double> table;

namespace Driver
{
  extern int no_of_errors;
  extern std::istream* input;
  void skip();
}
```

Ключевое слово ***extern*** используется для объявления переменных, чтобы гарантировать отсутствие множественных определений при включении файла ***dc.h*** в разные ***.c*** файлы. Соответствующие определения располагаются в том или ином ***.c*** файле.

За вычетом реального кода файл ***lexer.c*** будет выглядеть следующим образом:

```
// файл lexer.c:

#include "dc.h"
#include <iostream>
#include <cctype>
```

```
Lexer::Token_value Lexer::curr_tok;
double Lexer::number_value;
std::string Lexer::string_value;

Lexer::Token_value Lexer::get_token() {/* ... */}
```

Использование заголовочных файлов гарантирует, что каждое объявление из этих файлов будет помещено в *.c* файл, содержащий соответствующее определение. Например, при компиляции *lexer.c* компилятор повстречает объявление

```
namespace Lexer                    // из dc.h
{
  // ...
  Token_value get_token();
}
// ...
Lexer::Token_value Lexer::get_token() { /* ... */ }
```

Это, в свою очередь, гарантирует, что компилятором будут выявлены все случаи рассогласования типов. Например, если бы функция *get_token*() в объявлении возвращала бы значение типа *Token_value*, а в определении — значение типа *int*, то компиляция файла *lexer.c* завершилась бы с ошибкой несоответствия типов. Если отсутствует определение, ошибку обнаружит компоновщик, а если нет объявления — это обнаружит компилятор.

Файл *parser.c* будет выглядеть следующим образом:

```
// файл parser.c:

#include "dc.h"

double Parser::prim(bool get) {/* ... */}
double Parser::term(bool get) {/* ... */}
double Parser::expr(bool get) {/* ... */}
```

А вот содержимое файла table.c:

```
// файл table.c:

#include "dc.h"

std::map<std::string, double> table;
```

Таблица символов является просто глобальной переменной *table* стандартного библиотечного типа *map*. В программах реальных (больших) размеров загрязнение глобального пространства имен вызовет в конце концов проблемы. Но здесь я допустил эту небрежность намеренно, чтобы появился повод предупредить о возможных проблемах.

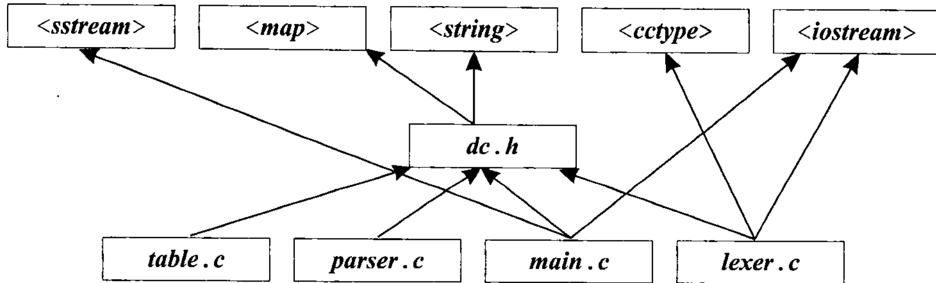
Наконец, файл *main.c* имеет следующий вид:

```
// файл main.c:

#include "dc.h"
#include <sstream>
#include <iostream>

int Driver::no_of_errors = 0;
std::istream* Driver::input= 0;
```

```
void Driver::skip() {/* ... */}

int main(int argc, char* argv[]) {/* ... */}
```

Функция *main*(), как стартовая функция программы, должна быть глобальной, и поэтому она не помещена ни в какое пространство имен.

Физическую структуру программы можно представить графически:



Заголовочные файлы в верхней части рисунка являются стандартными библиотечными заголовками. В большинстве случаев при анализе программ эти заголовочные файлы можно опустить, ибо про эту стабильную составляющую практически любой программы и так все известно заранее. Крошечные программы можно предельно упростить, поместив все директивы *#include* в один заголовочный файл.

Такой стиль физического сегментирования программ хорош для программ малых размеров, отдельные части которых не предполагается использовать отдельно. Если используются пространства имен, то логическая структура программы также отражается в *dc.h*. Если же пространства имен не используются, то структура становится туманной. В этом случае помогают комментарии.

Для программ больших размеров стиль сегментирования с единственным заголовочным файлом становится неработоспособным в обычных средах разработки, ориентированных на файлы. Изменения в единственном заголовочном файле вызывают перекомпиляцию всей программы, а работа с этим единственным файлом разных программистов чревата ошибками. По мере роста размеров программ их логическая структура катастрофически ухудшается, если разработка ведется без опоры на пространства имен и классы.

9.3.2. Несколько заголовочных файлов

Альтернативой является такая физическая организация программы, когда каждому логическому модулю соответствует свой заголовочный файл, содержащий все необходимые модулю объявления. Каждый *.c* файл располагает соответствующим заголовочным файлом, определяющим его интерфейс. Каждый *.c* файл директивами *#include* включает свой заголовочный файл и иные заголовочные файлы, определяющие то, что нужно модулю от других модулей. В этом случае физическая организация соответствует логической организации модуля. Интерфейс для пользователей модуля размещается в *.h* файле, интерфейс для реализации модуля — в файле с суффиксом *_impl.h*, а сама реализация модуля (функции, определения переменных и т.д.) сосредотачивается в *.c* файле. В результате модуль представляется тремя файлами. Например, интерфейс для пользователей модуля парсера располагается в файле *parser.h*:

```
// файл parser.h:

namespace Parser
{
  double expr (bool get) ;
}
```

Заголовочный файл реализации этого модуля помещается в заголовочный файл ***parser_impl.h***:

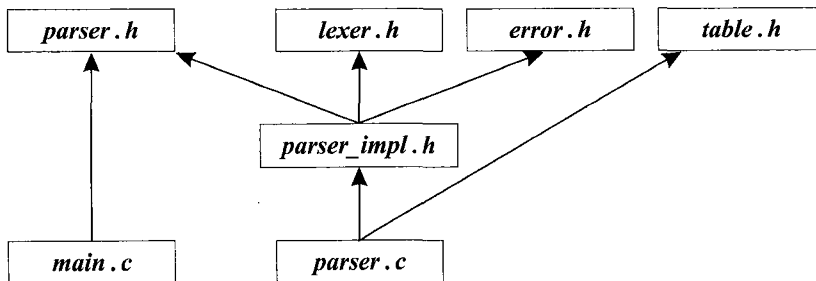
```
// файл parser_impl.h:

#include "parser.h"
#include "error.h"
#include "lexer.h"

namespace Parser              // интерфейс для реализации
{
  double prim (bool get) ;
  double term (bool get) ;
  double expr (bool get) ;

  using Lexer: :get_token;
  using Lexer: :curr_tok;
}
```

Сюда включен и заголовочный файл для пользователей ***parser.h***, чтобы компилятор имел возможность проверить согласованность объявлений (§9.3.1).

Реализация парсера располагается в файле ***parser.c***, включающем все необходимые ему заголовочные файлы:

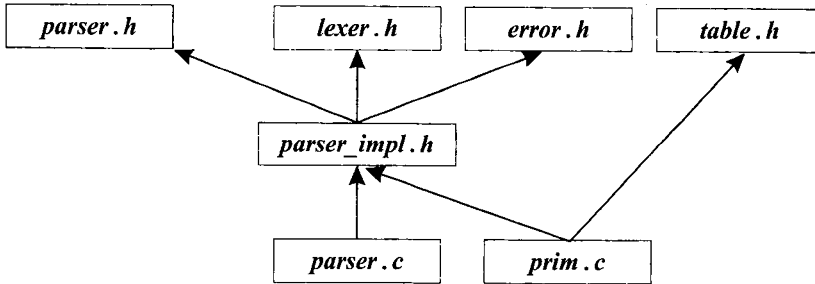
```
// файл parser.c:

#include "parser_impl.h"
#include "table.h"

double Parser: :prim (bool get)  {/* ... */}
double Parser: :term (bool get)  {/* ... */}
double Parser: :expr (bool get)  {/* ... */}
```

В графическом виде парсер и его использование управляющим модулем выглядит так:



Как и было задумано, здесь достигнуто близкое соответствие логической структуре, описанной ранее в §8.3.2. Для ее дальнейшего упрощения можно было бы по-

местить директиву *#include "table.h"* в файл ***parser_impl.h*** вместо файла ***parser.c***. Однако файл ***table.h*** не относится к общему интерфейсу функций синтаксического анализатора (парсера); скорее он нужен лишь реализации этих функций. Более точно, он нужен лишь функции ***prim()***, так что если уж мы на самом деле были бы настроены минимизировать зависимости, то стоило бы функцию ***prim()*** выделить в отдельный ***.c*** файл и только в него включать ***table.h***:



Столь тщательная проработка уместна, в первую очередь, для больших программ. Для них типично включение дополнительных заголовочных файлов только там, где это требуется отдельным функциям. Более того, для больших программ нередко используется несколько заголовочных файлов с суффиксом ***_impl.h***, поскольку разным подмножествам функций модуля требуются разные интерфейсы реализации.

Обратите внимание на то, что применение суффикса ***_impl.h*** не только не является стандартом, но даже и нельзя сказать, что сильно распространенным явлением; просто лично мне нравится такой стиль именования.

Итак, для чего же все-таки стоит возиться с более сложной схемой со многими заголовочными файлами? Ведь намного проще поместить все объявления в единственный заголовочный файл ***dc.h***.

Схема со *многими заголовочными файлами* отвечает масштабу модулей, во много раз больших нашего игрушечного парсера, и масштабу программ, намного больших нашего калькулятора. *Фундаментальная идея* сводится к *большей локализации*. Когда программист анализирует и модифицирует большую программу, ему важно иметь возможность *сфокусироваться на относительно небольшом фрагменте кода*. Схема организации программы со многими заголовочными файлами позволяет четко выявить, от чего зависит, например, модуль парсера, и *игнорировать остальные части программы*. Подход же с единственным заголовочным файлом заставляет нас в рамках каждого модуля просматривать все объявления и решать, какие из них необходимы. Факт состоит в том, что при этом приходится работать с кодом в условиях неполной информации, отталкиваясь от локальной перспективы. Подход же со многими заголовочными файлами позволяет нам в локальной перспективе успешно продвигаться «изнутри наружу» (from the inside out). Метод с единственным заголовком, как и любая другая организация программ с глобальным хранением информации, требует *подхода «сверху вниз»* (*top-down approach*) и заставляет нас бесконечно выяснять, что от чего зависит.

Более сильная локализация ведет, к тому же, и к меньшим временам компиляции. Эффект может быть весьма значительным. Я наблюдал уменьшение времени

компиляции в десятки раз из-за лучшего анализа зависимостей и лучшего применения заголовочных файлов.

9.3.2.1. Остальные модули калькулятора

Остальные модули калькулятора можно организовать аналогично модулю парсера. Однако они столь малы, что для них нет нужды в собственных ***_imlp.h*** файлах. Такие файлы нужны лишь логическим модулям, содержащим наборы функций с общим контекстом.

Обработчик ошибок сократился до набора типов исключений, так что для него не требуется ***error.c*** файл:

```
// файл error.h:

namespace Error
{
  struct Zero_divide { };
  struct Syntax_error
  {
    const char* p;
    Syntax_error (const char* q) {p = q; }
  };
}
```

Лексический анализатор обладает довольно большим и более запутанным интерфейсом:

```
// файл lexer.h:

#include <string>

namespace Lexer
{
  enum Token_value
  {
    NAME,       NUMBER,       END,
    PLUS='+',   MINUS='-',    MUL='*',     DIV='/',
    PRINT=';',  ASSIGN='=',   LP='(',      RP=')'
  };

  extern Token_value curr_tok;
  extern double number_value;
  extern std::string string_value;

  Token_value get_token ();
}
```

Реализация лексического анализатора кроме заголовка ***lexer.h*** зависит еще от ***error.h***, ***<iostream>***, ***driver.h*** и от функций, определяющих тип символов, объявленных в ***<cctype>***:

```
// файл lexer.c:

#include "lexer.h"
#include "error.h"
#include "driver.h"
```

```
#include <iostream>
#include <cctype>

Lexer::Token_value Lexer::curr_tok;
double Lexer::number_value;
std::string Lexer::string_value;

Lexer::Token_value Lexer::get_token() { /* ... */ }
```

Можно было бы выделить директиву *#include "error.h"* в отдельный файл с суффиксом *_impl.h*, но я счел это излишним для столь малой программы.

Как всегда, мы включаем файл с пользовательским интерфейсом модуля — в данном случае файл *lexer.h*, в файл реализации модуля, чтобы дать компилятору возможность проверить согласованность типов.

Причина, по которой мы вводим в программу файл *driver.h*, заключается в том, что функции *Lexer::get_token()* нужен доступ к средствам ввода:

```
// файл driver.h:

namespace Driver
{
  int no_of_errors;
  std::istream* input;
  void skip();
}
```

Таблица символов вполне самодостаточна, и стандартный заголовок <*map*> содержит все необходимое для ее эффективной реализации (в виде экземпляра шаблонного класса *map*):

```
// файл table.h:

#include <map>
#include <string>

extern std::map<std::string, double> table;
```

Поскольку каждый заголовочный файл может включаться в разные *.c* файлы, то нужно отделить объявление *table* от ее определения:

```
// файл table.c:

#include "table.h"

std::map<std::string, double> table;
```

По существу, управляющая программа зависит ото всего:

```
// файл main.c:

#include "parser.h"
#include "lexer.h"
#include "error.h"
#include "table.h"
#include "driver.h"
#include <sstream>

int main (int argc, char* argv[]) {/*...*/}
```

Для больших программ стоило бы реорганизовать структуру программы так, чтобы ее управляющая часть имела бы меньше прямых зависимостей. Часто также желательно минимизировать и код функции *main*(), поместив в нее лишь вызов управляющей функции, размещенной в своем собственном модуле. Это особенно важно для библиотечного кода, ибо тогда мы не можем полагаться на *main*() и должны быть готовы к тому, что вызовы могут последовать из самых разных функций (§9.6[8]).

9.3.2.2. Использование заголовочных файлов

Оптимальное для данной программы количество используемых заголовочных файлов зависит от многих факторов. Многие из этих факторов относятся даже не к языку C++, а к тому, как ведется работа с файлами в вашей системе. Например, если ваш редактор не допускает удобной параллельной работы со множеством файлов, то большое количество заголовочных файлов становится определенным препятствием в работе. Или, если ваш редактор на открытие и просмотр двадцати файлов по пятьдесят строк каждый требует больше времени, чем на ту же работу с единственным файлом из 1000 строк, то тут уж дважды подумаешь, прежде чем решишься применить схему организации небольшой программы в варианте со многими заголовками.

Предупредим, что дюжина собственных плюс ряд стандартных для конкретной реализации заголовков (могут исчисляться сотнями) вполне терпимы. Однако если вы сегментируете объявления в большой программе на логически минимальные заголовки (содержащие, например, объявление единственной структуры), то можете столкнуться с хаосом из сотен файлов даже для небольших проектов. Я нахожу это излишним.

Для больших проектов множественные заголовочные файлы неизбежны. В таких проектах сотни собственных (не считая стандартных) заголовков являются нормой. Настоящая проблема начинается, когда их число переваливает за тысячу. В масштабе таких проектов обсуждаемые здесь методики по-прежнему применимы, но это уже задача для Геракла. Запомните, что для настоящих больших программ единственный заголовочный файл — это не актуально. В них всегда присутствует много заголовочных файлов. Реальный выбор между двумя стилями организации заголовков переносится на меньшие части проектов.

На самом деле, обсуждаемые здесь два стиля организации заголовочных файлов вовсе не альтернативны друг другу. Скорее, они являются комплиментарными технологиями, выбор между которыми всегда стоит при разработке важных модулей, и вопрос о таком выборе возникает снова и снова по мере эволюции программной системы. Важно понимать, что один и тот же интерфейс не может быть оптимальным для всех. Обычно стоит различать интерфейс для пользователей и интерфейс для разработчиков. Дополнительно, часто в больших системах большинству пользователей предоставляется несколько упрощенный интерфейс, в то время как дополнительные возможности предоставляются пользователям-экспертам. Экспертный интерфейс («полный интерфейс») обычно директивами *#include* включает намного больше средств, чем хотел бы знать обычный пользователь системы. На практике, обычный пользовательский интерфейс получается после удаления из экспертного интерфейса как раз таких средств (и соответственно директив *#include*), которые неизвестны рядовому пользователю. Термин «рядовой пользова-

тель» не является уничижительным. В тех областях, где я не обязан быть экспертом, я предпочитаю быть рядовым пользователем — это помогает избегать ненужных трудностей.

9.3.3. Защита от повторных включений

Целью применения множественных заголовочных файлов является построение согласованных и самодостаточных модулей. С точки зрения программы многие объявления, необходимые для обеспечения самодостаточности отдельных модулей, в целом избыточны. В больших программах такая избыточность может приводить к ошибкам, ибо определения классов или встраиваемых функций могут через директивы ***#include*** (в том числе и вложенные) попасть в отдельные единицы трансляции более одного раза (§9.2.3).

У нас есть две возможности:

1. Реорганизовать программу для устранения избыточности, или
2. Найти способ безопасного повторного включения заголовков.

Первый подход — именно он ведет к нашей последней версии калькулятора — весьма утомителен и непрактичен в случае реально больших программ. Кроме того, избыточность ведь делает отдельные части программ более осмысленными.

Конечно, выигрыш от тщательного анализа избыточности и результирующего упрощения программы может быть значительным как с логической точки зрения, так и в плане сокращения времени компиляции. Однако такой анализ редко когда бывает полным, так что в любом случае требуется найти метод преодоления негативных последствий множественного включения заголовков. Желательно, чтобы этот метод можно было применять систематически, ибо заранее неизвестна полнота анализа, проводимого пользователем.

Традиционным решением проблемы является использование следующей комбинации директив условной компиляции — ***#ifndef***, ***#define*** и ***#endif***:

```
// файл error.h:

#ifndef CALC_EROR_H
#define CALC_EROR_H

namespace Error
{
  // ...
}

#endif                    // CALC_ERROR_H
```

Содержимое заголовочного файла между ***#ifndef*** и ***#endif*** игнорируется компилятором, если макроконстанта ***CALC_ERROR_H*** определена. Таким образом, когда заголовочный файл ***error.h*** первый раз встречается при компиляции, его содержимое вычитывается полностью, а константа ***CALC_ERROR_H*** становится определенной (в результате действия директивы ***#define***). Но если компилятор в пределах той же самой единицы трансляции встретит ***error.h*** еще раз, его содержимое будет проигнорировано. Этот метод, хоть он и опирается на трюкачество с макросами, реально работает и широко распространен в мире программирования на C и C++. Все стандартные заголовочные файлы используют этот метод.

Так как заголовочные файлы включаются в произвольном контексте, то следует учитывать возможность конфликта имен, из-за чего для макроконстант, контролирующих множественные включения, рекомендуется выбирать длинные и «безобразно» выглядящие имена.

Как только люди привыкают к заголовочным файлам и получают метод избавления от множественных включений, они тотчас же приступают к неумеренному наращиванию количества прямо или косвенно включаемых заголовочных файлов. Даже в реализациях с оптимизированной обработкой заголовков, такое нежелательно, ибо может сильно увеличивать время компиляции и засоряет глобальную область видимости объявлениями и макросами. Последнее может оказать непредсказуемое влияние на смысл программы. Заголовочные файлы следует использовать лишь тогда, когда в этом есть действительная необходимость.

9.4. Программы

Программа состоит из множества раздельно компилируемых единиц, собираемых воедино компоновщиком. Каждая функция, объект, тип и т.п. должен иметь уникальное определение в рамках этого множества (§4.9, §9.2.3). В программе должна быть единственная функция *main*() (§3.2). Вычислительная активность программы начинается в функции *main*() и завершается по выходу из этой функции. Возвращаемое этой функцией целое значение передается программному загрузчику в качестве результата работы программы.

Рассмотренная простая схема получает дополнительные детали в программах, содержащих глобальные переменные (§10.4.9), или в программах, генерирующих неперехватываемые исключения (§14.7).

9.4.1. Инициализация нелокальных переменных

В принципе, любая переменная, определяемая вне функции (в глобальном пространстве, в именованном пространстве имен, в качестве статического члена класса), инициализируется до вызова *main*(). Порядок инициализации таких переменных соответствует порядку их определения в единице трансляции (§10.4.9). Если явные инициализаторы отсутствуют, то переменные инициализируются умолчательными значениями, характерными для их типов данных (§10.4.2). Умолчательное значение для встроенных типов и перечислений равно ***нулю***. Например:

```
double x=2;              // нелокальные переменные
double y;
double sqx=sqrt(x+y);
```

В этом примере нелокальные переменные *x* и *y* инициализируются раньше, чем переменная *sqx*, так что при инициализации последней вычисляется *sqrt*(*2*).

Порядок инициализации глобальных переменных, определяемых в разных единицах трансляции, стандартом языка C++ не фиксируется. Поэтому крайне неразумно строить код, зависящий от порядка инициализации глобальных переменных. Кроме того, невозможно перехватить исключение, сгенерированное инициализатором глобальной переменной (§14.7). Использование глобальных переменных вообще желательно минимизировать, в особенности тех, что требуют сложной инициализации.

В принципе, существует ряд приемов, позволяющих фиксировать порядок инициализации глобальных переменных, определенных в разных единицах трансляции. Однако они не являются ни эффективными, ни переносимыми. Например, динамические библиотеки плохо переживают зависимость от таких глобальных переменных.

Хорошей альтернативой глобальным переменным могут служить функции, возвращающие ссылку. Например:

```
int& use_count()
{
  static int uc = 0;
  return uc;
}
```

Вызов *use_count*() действует как глобальная переменная, инициализация которой происходит при самом первом вызове (§5.5, §7.1.2). Например:

```
void f()
{
  cout<< ++use_count();     // увеличить значение и вывести его
  // ...
}
```

Инициализация нелокальных переменных (статически распределенных) зависит от механизма запуска программ, реализованного в конкретной системе. Гарантируется правильность работы такого механизма при вызове функции *main*(). Отсюда следует, что в общем случае невозможно обеспечить правильность инициализации нелокальных переменнных в C++-коде, если сам этот код предназначен для работы в программах, написанных на иных языках программирования.

Отметим, что переменные, инициализируемые константными выражениями (§C.5) не могут зависеть от значения объектов из других единиц трансляции и не требуют инициализации на этапе выполнения. Такие переменные можно безопасно использовать во всех случаях.

9.4.1.1. Завершение выполнения программы

Можно завершить работу одним из следующих способов:

- возвратом из функции *main*();
- вызовом *exit*();
- вызовом *abort*();
- генерацией неперехватываемого исключения.

Дополнительно существуют нерекомендуемые, зависящие от реализации способы доведения программы до краха.

Если выход из программы осуществляется вызовом стандартной библиотечной функции *exit*(), то вызываются деструкторы для всех статически сконструированных объектов (§10.4.9, §10.2.4). В то же время, если прекращение работы программы достигается вызовом стандартной библиотечной функции *abort*(), то этого не происходит. Отсюда следует, что при использовании функции *exit*() мгновенного прекращения работы программы не происходит. Более того, вызов *exit*() из дест-

руктора вообще может привести к бесконечной рекурсии. Функция *exit*() объявляется следующим образом

```
void exit(int);
```

Как и у функции *main*(), возврат *exit*() адресован системе в качестве результата работы программы. Нуль означает успешное завершение.

При вызове *exit*() не будут вызваны деструкторы локальных переменных во всех функциях вверх по цепочке имевших место функциональных вызовов. Генерация и перехват исключений гарантируют корректное уничтожение локальных объектов (§14.4.7). Вызов же *exit*() не позволяет корректно отработать всем функциям из цепочки вызовов. Поэтому лучше не ломать контекст вызова функций и просто сгенерировать исключение, а вопрос о том, что делать дальше, оставить обработчикам.

В языках C и C++ стандартная библиотечная функция *atexit*() позволяет вызвать некоторый код по выходу из программы. Например:

```
void my_cleanup();

void somewhere()
{
 if(atexit(&my_cleanup)==0)
 {
  // my_cleanup будет вызвана при нормальном завершении
 }
 else
 {
  // oops: слишком много функций atexit
 }
}
```

Это уже сильно напоминает автоматический вызов деструкторов глобальных переменных по выходу из программы (§10.4.9, §10.2.4). Обратите внимание, что функция-аргумент *atexit*() сама не может иметь аргументов и возвращаемого значения. От конкретной реализации зависит предельное количество функций *atexit*(); при превышении предела *atexit*() возвращает нуль. Это делает *atexit*() менее полезной, чем могло показаться на первый взгляд.

Деструкторы статически размещенных объектов (глобальных, §10.4.9; локальных в функциях с модификатором *static*, §7.1.2; определенных с модификатором *static* в классах, §10.2.4), созданных до вызова *atexit*(*f*), вызываются после вызова *f*(). Деструкторы же таких объектов, созданных после вызова *atexit*(), вызываются до вызова *f*().

Функции *exit*(), *abort*() и *atexit*() объявлены в *<cstdlib>*.

9.5. Советы

1. Используйте заголовочные файлы для представления интерфейсов и логической структуры; §9.1, §9.3.2.
2. Включайте заголовочный файл в исходный файл, реализующий объявленные функции; §9.3.1.
3. Не определяйте глобальные сущности с одинаковыми именами, но с разным (пусть и похожим) смыслом в разных единицах трансляции; §9.2.
4. Избегайте определений невстраиваемых функций в заголовочных файлах; §9.2.1.
5. Используйте директивы *#include* только в глобальной области или в пространствах имен; §9.2.1.
6. С помощью *#include* включайте только полные объявления; §9.2.1.
7. Реализуйте защиту от лишних включений; §9.3.3.
8. Заключайте заголовочные файлы языка C в пространства имен во избежание конфликта глобальных имен; §8.2.9.1, §9.2.2.
9. Делайте заголовочные файлы самодостаточными; §9.2.3.
10. Различайте пользовательские интерфейсы и интерфейсы для разработчиков; §9.3.2.
11. Различайте обычные пользовательские интерфейсы и интерфейсы для пользователей-экспертов; §9.3.2.
12. Избегайте нелокальных объектов с инициализацией на этапе выполнения, предназначенных для использования в качестве фрагментов программ, написанных не на C++; §9.4.1.

9.6. Упражнения

1. (*2) Найдите, где в вашей системе находятся заголовочные файлы стандартной библиотеки. Просмотрите список их имен. Есть ли среди них нестандартные заголовочные файлы? Могут ли нестандартные заголовки включаться с помощью <>?
2. (*2) Где хранятся заголовочные файлы нестандартных «фундаментальных» библиотек вашей системы?
3. (*2.5) Напишите программу, которая читает исходные файлы на C++ и выводит имена файлов, включенных в них директивой *#include*. В результирующем списке примените отступы для наглядного показа информации о том, какие файлы включаются в тот или иной исходный файл. Опробуйте эту программу на реальных исходных файлах (и получите информацию об объеме включаемых файлов).
4. (*3) Модифицируйте программу из предыдущего упражнения, чтобы она выводила количество строк-комментариев, количество строк без комментариев, количество слов (разделенных пробельными символами) в каждом включаемом файле.

5. (*2.5) Разработайте внешнее средство, которое проверяет перед компиляцией каждый исходный файл на предмет существования повторных включений заголовков и устраняет повторы. Опробуйте это средство на практике и сравните его со способами контроля повторных включений, рассмотренных в §9.3.3. Дает ли такое средство какие-либо дополнительные преимущества на этапе выполнения на вашей системе?
6. (*3) Как в вашей системе реализуется динамическая компоновка? Какие ограничения накладываются на динамически компонуемый код? Что дополнительно требуется от кода, чтобы он мог быть скомпонован динамически?
7. (*3) Откройте и прочтите 100 файлов по 1500 символов каждый. Откройте и прочтите один файл, содержащий 150000 символов (см. пример в §21.5.1). Есть ли разница во времени выполнения задач? Каково наибольшее число файлов, которые можно одновременно открыть в вашей системе? Рассмотрите этот вопрос касательно использования включаемых файлов.
8. (*2) Модифицируйте нашу версию калькулятора так, чтобы его можно было вызывать из ***main*** `()`, либо из любой другой функции (в виде обычного функционального вызова).
9. (*2) Нарисуйте диаграмму зависимостей модулей (§9.3.2) для версии калькулятора, которая использует ***error*** `()` вместо исключений (§8.2.2).

Часть II

Механизмы абстракции

Здесь описываются средства языка C++ для определения и использования новых типов. Представлены методики объектно-ориентированного и обобщенного стилей программирования.

Главы

10. Классы

11. Перегрузка операций

12. Наследование классов

13. Шаблоны

14. Обработка исключений

15. Иерархии классов

«... нет дела, коего устройство было бы труднее, ведение опаснее, а успех сомнительнее, нежели замена старых порядков новыми. Кто бы ни выступал с подобным начинанием, его ожидает враждебность тех, кому выгодны старые порядки, и холодность тех, кому выгодны новые ...»

— Николо Макиавелли («Государь» §vi)

10

Классы

Эти типы совсем не «абстрактные»;
они столь же реальны как ***int*** *или* ***float****.*
— Дуг МакИлрой

Концепции и классы — члены класса — управление доступом — конструкторы — *статические* члены класса — копирование по умолчанию — *константные* функции-члены — ***this*** — *структуры* — определение функции в классе — конкретные классы — функции-члены и функции поддержки — перегруженные операции — использование конкретных классов — деструкторы — конструкторы по умолчанию — локальные переменные — копирование, определяемое пользователем — ***new*** и ***delete*** — классовые объекты как члены класса — массивы — статическая память — временные объекты — объединения — советы — упражнения.

10.1. Введение

Концепция классов языка C++ нацелена на то, чтобы снабдить программиста средствами создания новых типов данных, пользоваться которыми удобно в той же мере, что и встроенными типами. Кроме того, концепции производных классов (глава 12) и шаблонов (глава 13) позволяют программисту синтаксическим образом извлечь практическую выгоду из определенного рода отношений между классами.

Тип есть конкретное представление концепции. Например, встроенный тип ***float*** вместе с операциями +, –, * и т.д. представляет собой конкретную (приближенную) реализацию математической концепции вещественных чисел. *Класс* — это *тип*, *определяемый пользователем* (программистом). Мы создаем новые типы, чтобы точнее отражать концепции, не представимые встроенными типами. Например, для телефонной программы может пригодиться тип ***Trunk_line*** (магистральная междугородняя телефонная линия), для видеоигры — тип ***Explosion*** (взрыв), а для текстового процессора — тип ***list<Paragraph>*** (список параграфов, то есть абзацев). Программы, в которых типы близки концепциям предметной области задачи, и понимать легче, и легче сопровождать (в том числе модифицировать) в течении всего срока их эксплуатации.

Умело подобранный набор пользовательских типов делает программу более компактной и более выразительной. Он также позволяет выполнять тщательный анализ кода. В частности, компилятор может при этом надежно выявлять случаи недопустимого использования объектов указанных типов, которые в противном случае не были бы выявлены иначе, как с помощью процесса массированной и длительной отладки.

Фундаментальная идея, которой нужно следовать при построении нового типа, заключается в том, чтобы четко отделить случайные детали реализации (например, расположение в памяти отдельных частей объектов этого типа) от таких свойств, которые обеспечивают удобное и безошибочное использование типа в клиентском коде (например, полный набор функций для доступа к данным). Подобное разделение лучше всего реализуется путем предоставления доступа к структурам данных и другим внутренним подробностям исключительно через специально предназначенный для этого интерфейс.

В настоящей главе основное внимание фокусируется на относительно простых конкретных классах, которые с логической точки зрения слабо отличаются от встроенных типов. В идеале, такие типы должны отличаться от встроенных типов не в плане использования клиентским кодом, а только способами их создания.

10.2. Классы

Класс — этот тип, определяемый пользователем. В данном разделе рассматриваются основные средства определения классов, создания объектов классов и манипулирования объектами классов.

10.2.1. Функции-члены

Рассмотрим реализацию концепции даты с помощью *структуры* ***Date***, содержащей данные, и набора функций для манипулирования переменными этого типа:

```
struct Date                                    // представление
{
  int d, m, y;
};

void init_date(Date& d, int, int, int);       // инициализация d
void add_year(Date& d, int n);                // прибавить n лет к d
void add_month(Date& d, int n);               // прибавить n месяцев к d
void add_day(Date& d, int n);                 // прибавить n дней к d
```

Явной связи между структурой и набором функций нет. Такую связь можно выразить явно, объявив функции в качестве членов структуры:

```
struct Date
{
  int d, m, y;
  void init(int dd, int mm, int yy);          // инициализация
  void add_year(int n);                       // прибавить n лет
  void add_month(int n);                      // прибавить n месяцев
  void add_day(int n);                        // прибавить n дней
};
```

Функции, объявленные в теле определения класса (а ***структура*** это один из видов класса; §10.2.8), называются *функциями-членами* (*member functions*)[1] и могут вызываться только от имени переменных этого типа, используя стандартный синтаксис доступа к членам структуры. Например:

```
Date my_birhday;

void f()
{
  Date today;
  today.init(16,10,1996);
  my_birthday.init(30,12,1950);
  Date tomorrow = today;
  tomorrow.add_day(1);
  // ...
}
```

Так как различные структуры могут иметь функции-члены с одинаковыми именами, мы должны явно указывать имя структуры в определении функций-членов:

```
void Date::init(int dd, int mm, int yy)
{
  d = dd;
  m = mm;
  y = yy;
}
```

В теле функций-членов имена членов класса можно использовать без явных ссылок на объект класса. Это означает, что имена ссылаются на поля объекта, для которого функция-член вызвана. К примеру, если функция-член ***Date::init()*** вызвана для объекта ***today***, то ***m=mm*** означает ***today.m=mm***, а если эта функция-член вызвана для объекта ***my_birthday***, то ***m=mm*** означает ***my_birthday.m=mm***. В процессе своей работы функция-член всегда «знает», для какого объекта она вызвана.

Конструкция

```
class X { ... };
```

называется *определением класса* (*class definition*), так как с ее помощью определяется новый тип данных. По историческим причинам часто определение класса называют *объявлением класса* (*class declaration*). Как и объявления, не являющиеся определениями, определение класса может без нарушения правила одного определения (§9.2.3) присутствовать в разных исходных файлах программы, будучи помещенным в них директивой ***#include***.

10.2.2. Управление режимом доступа

Объявление ***Date*** из предыдущего раздела предоставляет набор функций для манипулирования объектами этого типа. Однако оно не фиксирует того факта, что лишь эти функции знают точное устройство объектов типа и только через них осу-

[1] Постепенно, особенно в русскоязычной литературе, вместо термина «функция-член» все чаще применяется термин «метод класса» (англ. термин — class method). — *Прим. ред.*

ществляется прямой доступ к этим объектам. Чтобы выразить указанное ограничение явным образом, применим ключевое слово ***class*** вместо ***struct***:

```
class Date
{
  int d, m, y;

public:
  void init(int dd, int mm, int yyy);   // инициализация
  void add_year(int n);                 // прибавить n лет
  void add_month(int n);                // прибавить n месяцев
  void add_day(int n);                  // прибавить n дней
};
```

Метка ***public*** делит тело класса на две части. Имена в первой части (*закрытой — private*) могут использоваться лишь функциями-членами. Вторая же часть — *открытая* (*public*) и она формирует открытый интерфейс к объектам класса. *Структуры* есть по сути те же *классы*, но с открытым режимом доступа по умолчанию (§10.2.8); функции-члены здесь определяются и используются одинаково. Например:

```
inline void Date::add_year(int n)
{
  y += n;
}
```

Обычные же функции (не функции-члены) теперь лишены возможности обращаться к закрытым полям класса напрямую. Например:

```
void timewarp(Date& d)
{
  d.y -= 200;                  // error: Date::y имеет режим private
}
```

Из ограничения доступа к внутренней структуре класса явно предназначенным для этого интерфейсом вытекает ряд преимуществ. Например, любая ошибка, приводящая к неверным данным в объекте типа ***Date*** (например, 36 декабря 1985 года), является следствием неверного кода в одной из функций-членов. Это означает, что локализация ошибки (первая стадия отладки) может быть выполнена на ранней стадии разработки программы (чуть ли не до ее запуска). Более общее положение состоит в том, что любое изменение поведения типа ***Date*** может и должно осуществляться изменением кода функций-членов. Например, любое усовершенствование внутренней структуры данных класса требует лишь изменения функций-членов с тем, чтобы воспользоваться новыми возможностями. Код же клиента, опирающегося на открытый интерфейс класса, остается при этом неизменным (может потребоваться его перекомпиляция). Еще одно достоинство состоит в том, что потенциальному пользователю класса для начала работы с новым типом данных нужно изучить лишь работу с открытым классовым интерфейсом.

Защита закрытых данных класса опирается на защиту имен членов класса. Такую защиту можно обойти с помощью манипуляции адресами и явного преобразования типов (то есть обманным путем). Защита C++ есть *защита от непреднамеренных ошибок*, а не от умышленного нарушения правил. Только аппаратная защита могла бы поставить надежный барьер на пути злонамеренных фокусов с универ-

сальным языком высокого уровня, что, однако, трудновыполнимо в реальных системах.

Из-за закрытия данных в классе ***Date*** мы вынуждены были ввести функцию ***init***() , с помощью которой можно установить конкретное значение объекта класса.

10.2.3. Конструкторы

Использование функций типа ***init***() для инициализации объектов класса и неэлегантно и чревато ошибками. Действительно, поскольку нигде прямо не указано, что объект класса нужно инициализировать таким образом, программист может забыть об этом или, наоборот, дважды вызвать функцию ***init***() (часто с одинаковыми разрушительными последствиями). В итоге, лучше дать возможность программисту объявлять специальные функции-члены, явно предназначенные именно для инициализации объектов. Так как такие функции участвуют в создании (конструировании) объектов, их принято называть *конструкторами* (*constructors*). Конструкторы выделяются тем, что *их имена совпадают с именем класса*. Например:

```
class Date
{
  // ...
  Date(int, int, int);                // конструктор
};
```

Если класс имеет конструкторы, то любой объект класса инициализируется вызовом конструктора (§10.4.3), причем если конструктор имеет аргументы, то ему их требуется передать:

```
Date today = Date(23,6,1983);
Date xmas(25,12,1990);                // сокращенная форма
Date my_birthday;                     // error: отсутствует инициализация
Date release1_0(10,12);               // error: отсутствует третий аргумент
```

Часто бывает полезным иметь несколько способов инициализации классовых объектов. Этого можно достичь, определив в классе несколько конструкторов. Например:

```
class Date
{
  int d, m, y;
public:
  // ...
  Date(int, int, int);                // день, месяц, год
  Date(int, int);                     // день, месяц, текущий год
  Date(int);                          // день, текущие месяц и год
  Date();                             // дата по умолчанию: сегодня
  Date(const char*);                  // дата в строковом представлении
};
```

Конструкторы подчиняются тем же самым правилам перегрузки, что и обычные функции (§7.4). До тех пор, пока конструкторы существенно отличаются друг от друга набором аргументов, у компилятора есть возможность выбрать правильный вариант:

```
Date today(4);
Date july4("July 4, 1983");
Date guy("5 Nov");
Date now;            // умолчательная инициализация текущей датой
```

Чрезмерное увеличение числа конструкторов, как в показанном примере класса ***Date***, является типичным явлением. В процессе проектирования класса у программиста возникает искушение на всякий случай добавить любые средства просто потому, что они могут кому-нибудь понадобиться. Большие умственные усилия требуются, чтобы четко ограничить набор средств действительно необходимым минимумом. Эти дополнительные интеллектуальные усилия приводят в типичном случае к более ясным и более компактным программам. Одним из способов уменьшения количества необходимых функций является применение «аргументов по умолчанию» (§7.5). Для класса ***Date*** можно предусмотреть умолчательные значения аргументов, трактуемые как «сегодня» (***today***):

```
class Date
{
  int d, m, y;
public:
  Date(int dd =0, int mm =0, int yy =0);
  //...
};

Date::Date(int dd, int mm, int yy)
{
  d = dd?dd:today.d;
  m = mm?mm:today.m;
  y = yy?yy:today.y;
  // проверка допустимости даты
}
```

Для умолчательных значений аргументов нужно предусмотреть, чтобы они не совпадали с возможными «обычными» значениями аргументов. Для *дней* и *месяцев* это достигнуто с очевидностью (не существует нулевых дней и месяцев), но для *года* это также имеет место (хотя и не так очевидно), ибо в европейском календаре нет нулевого года; *первый год* от Рождества Христова следует сразу же за первым годом до Рождества Христова (*минус первый год*).

10.2.4. Статические члены

Удобство умолчательных аргументов в классе ***Date*** достигнуто за счет следующей скрытой проблемы. Класс ***Date*** теперь зависит от глобальной переменной ***today***. В результате безошибочное использование класса ***Date*** возможно лишь в контексте определения и корректного использования глобальной переменной ***today*** всеми частями программы. Такого рода ограничение делает класс бесполезным вне контекста, в котором он был изначально написан. Пользователи получают множество неприятных сюрпризов, когда они пытаются воспользоваться контекстно-зависимыми классами, а сопровождение программ становится слишком сложным. Может, «одна небольшая глобальная переменная» и не повредит, но вообще такой стиль программирования приводит к коду,

бесполезному для всех, кроме написавшего его программиста. Этого следует избегать.

К счастью, имеется возможность и удобства умолчательных значений сохранить, и глобальных переменных избежать. Переменная, являющаяся членом класса, но не частью объекта класса, называется *статическим членом* (*static member*). Она всегда существует в *единственном экземпляре*, в отличие от обычных членов класса, дублирующихся во всех классовых объектах (§С.9). Аналогично, функция, которой нужен доступ к членам класса, но не требуется ее вызов для конкретных объектов класса, называется *статической функцией-членом*.

Вот переработанный вариант класса ***Date***, сохраняющий семантику умолчательного конструирования, но не опирающийся при этом на глобальные переменные:

```
class Date
{
  int d, m, y;
  static Date default_date;

public:
  Date (int dd=0, int mm=0, int yy =0) ;
  // ...
  static void set_default (int, int, int) ;
};
```

Теперь мы можем определить конструктор класса ***Date*** так, чтобы использовать ***default_date***:

```
Date: :Date (int dd, int mm, int yy)
{
  d = dd ? dd: default_date.d;
  m = mm ? mm: default_date.m;
  y = yy ? yy: default_date.y;
  // проверка допустимости даты
}
```

Используя ***set_default***(), мы в любое время можем изменить умолчательное значение даты. К статическому члену можно обращаться так же, как и к любому другому. Дополнительно, к нему можно обращаться без указания классового объекта. Вместо этого следует квалифицировать его имя именем класса. Например:

```
void f()
{
  Date: :set_default (4,5,19445) ;          // обращение к статической функции
                                            // set_default() класса Date
}
```

Статические члены — как данные, так и методы — *должны быть определены*. Ключевое слово ***static*** при этом повторять не надо. Вот пример:

```
Date Date: :default_date (16,12,1770) ;      //определяем Date::default_date

void Date: :set_default (int d, int m, int y)   //определяем Date::set_default()
{
  default_date = Date (d,m,y) ;  //присваиваем новое значение умолчательной дате
}
```

Теперь умолчательная дата совпадает с днем рождения Бетховена (пока кто-то не задаст иное умолчательное значение).

Заметим, что выражение *Date*() служит иным обозначением умолчательного значения *Date*::*default_date*. Например:

```
Date copy_of_default_date = Date();
```

Следовательно, нам не нужна отдельная функция для чтения умолчательной даты.

10.2.5. Копирование объектов класса

Возможность копирования классовых объектов доступна изначально. В частности, объект класса можно инициализировать копией другого объекта того же класса; это можно делать и в случае, когда определены конструкторы. Например:

```
Date d = today;                          // инициализация копированием
```

По умолчанию, копия объекта класса содержит копии всех полей объекта. Если это не то, что нужно, следует явно определить *копирующий конструктор* (*copy constructor*) вида *X*::*X*(*const X*&) (подробно обсуждается далее в §10.4.4.1).

Аналогично, классовые объекты можно копировать с помощью *операции присваивания* (*assignment operator*). Например:

```
void f(Date& d)
{
  d = today;
}
```

И снова умолчательной семантикой операции является *почленное копирование*. Если это не подходит для некоторого класса *X*, то можно определить специфическую версию операции присваивания именно для этого класса (§10.4.4.1).

10.2.6. Константные функции-члены

В определенном нами классе *Date* имеются функции-члены, позволяющие задавать и изменять значения объектов этого класса. К сожалению, пока что отсутствует способ узнать эти значения. Проблема легко устраняется введением функций для чтения дня, месяца и года:

```
class Date
{
  int d, m, y;

public:
  int day() const {return d; }
  int month() const {return m; }
  int year() const;
  // ...
};
```

Обратите внимание на ключевое слово *const* после круглых скобок в объявлениях (определениях) функций. Оно означает, что эти *функции не изменяют состояния объектов класса Date* (*доступ по чтению*). Естественно, что компиляторы обнаруживают случайные попытки нарушить это обещание. Например:

```
inline int Date::year() const
{
  return y++;    // error: попытка изменить состояние объекта в константной функции
}
```

Когда константная функция-член определяется вне тела определения класса, требуется повторять ключевое слово ***const***:

```
inline int Date::year() const    // так правильно
{
  return y;
}

inline int Date::year()          // а так ошибка: отсутствует ключевое слово const
{
  return y;
}
```

Другими словами, *суффикс* ***const*** является неотъемлемой частью типа ***Date::day()***, ***Date::month()*** и ***Date::year()***.

Константные функции-члены могут вызываться для ***const***- и не ***const***-объектов классов, в то время как неконстантные функции-члены — лишь для не ***const***-объектов. Например:

```
void f(Date& d, const Date& cd)
{
  int i = d.year();       // ok
  d.add_year(1);          // ok
  int j = cd.year();      // ok
  cd.add_year(1);         // error: нельзя изменять значение константы cd
}
```

10.2.7. Ссылки на себя

Функции ***add_year()***, ***add_month()*** и ***add_day()***, призванные изменять значения объектов типа ***Date***, не имели возврата (указан как ***void***). Для подобных логически связанных функций было бы желательно иметь возможность их последовательного использования в виде компактной цепочки вызовов. Например, хотелось бы иметь возможность писать что-то вроде

```
void f(Date& d)
{
  // ...
  d.add_day(1).add_month(1).add_year(1);
  // ...
}
```

чтобы в рамках единственного выражения изменить значения дня, месяца и года. Для этого каждая из функций *должна возвращать ссылку на тип* ***Date***:

```
class Date
{
  // ...
  Date& add_year(int n);     // прибавить n лет
```

```
    Date& add_month(int n);     // прибавить n месяцев
    Date& add_day(int n);       // прибавить n дней
};
```

Каждая нестатическая функция класса знает объект, для которого она вызывается, и может ссылаться на него явным образом:

```
Date& Date::add_year(int n)
{
  if(d==29 && m==2 && !leapyear(y+n))      // не забудьте о February 29
  {
    d = 1;
    m = 3;
  }

  y += n;
  return *this;
}
```

Выражение ***this** как раз и означает такой объект. В языке Simula это же обозначается как ***THIS***, а в языке Smalltalk — как ***self***.

В нестатической функции-члене ключевое слово ***this*** означает *указатель на объект, для которого функция вызвана*. В неконстантной функции-члене класса *X* тип ***this*** есть *X**. Однако ж, ***this*** *не является обычной переменной — невозможно взять ее адрес и нельзя ей ничего присвоить*. В константной функции-члене класса *X* тип ***this*** есть ***const*** *X** для предотвращения изменения самого объекта (см. также §5.4.1).

По большей части ***this*** используется неявным образом. В частности, каждое обращение к нестатическому члену внутри класса опирается на неявное применение ***this*** для доступа к полям классового объекта. Например, можно написать следующее эквивалентное (но утомительное) определение функции ***add_year***():

```
Date& Date::add_year(int n)
{
  // не забудьте о February 29
  if(this->d==29 && this->m==2 && !leapyear(this->y+n))
  {
    this->d = 1;
    this->m = 3;
  }

  this->y += n;
  return *this;
}
```

В операциях со связными списками ***this*** применяется явным образом (§24.3.7.4).

10.2.7.1. Физическое и логическое постоянство

Иногда так бывает, что *константной* по логике функции-члену все-таки требуется изменить значение поля данных класса. С точки зрения пользователя функция ничего в состоянии объекта не меняет. А на самом деле она изменяет отдельные детали, пользователю недоступные. Такое явление называется *логическим постоянством* (*logical constness*). Например, класс ***Date*** мог бы возвращать строковое представление даты, которое пользователь использовал бы для вывода. Конструирование та-

кого представления может оказаться довольно дорогой (затратной) операцией. Поэтому имеет смысл хранить актуальную копию строки с тем, чтобы при последовательных запросах возвращать эту копию в случаях, когда дата не изменялась с момента последнего обращения. *Кэширование* (*caching*) данных чаще применяется для более сложных структур, но мы сейчас рассмотрим как все это работает на примере ***Date***:

```
class Date
{
  bool cache_valid;
  string cache;
  void compute_cache_value () ;          // заполнить кэш
  // ...

public:
  // ...
  string string_rep () const;            // строковое представление
};
```

С точки зрения пользователя, вызов ***string_rep***() не меняет состояния ***Date***, так что естественно, что это константная функция-член класса ***Date***. С другой стороны, любой кэш нужно хотя бы один раз проинициализировать до того, как он будет использоваться на чтение. Достигнуть этого можно, например, следующим образом (применяя грубую силу — ***brute force***):

```
string Date::string_rep () const
{
  if(cache_valid == false)
  {
    Date* th = const_cast<Date*> (this) ;    // снимаем const
    th->compute_cach_value () ;
    th->cache_valid = true;
  }
  return cache;
}
```

Мы здесь применяем операцию ***const_cast*** (§15.4.2.1) для того, чтобы вместо ***this*** получить указатель типа ***Date****. Это не только не элегантно, но может и не сработать, когда объект класса изначально объявлен константой. Например:

```
Date d1;
const Date d2;
string s1 = d1.string_rep () ;
string s2 = d2.string_rep () ;          // неопределенное поведение
```

Для объекта ***d1*** все прекрасно работает. Но объект ***d2*** объявлен константой и конкретная реализация C++ может включить дополнительные защитные механизмы, гарантирующие неизменность объекта. Из-за этого нельзя гарантировать, что ***d2.string_rep***() во всех реализациях будет иметь одинаковое и предсказуемое поведение.

10.2.7.2. Ключевое слово mutable

Можно избежать операции приведения ***const_cast*** и последующей зависимости от деталей реализации С++, если пометить кэшируемые данные класса ***Date*** ключевым словом ***mutable***:

```
class Date
{
  mutable bool cache_valid;
  mutable string cache;
  void compute_cache_value() const;      // заполнить (mutable) кэш
  // ...

public:
  // ...
  string string_rep() const;             // строковое представление
};
```

Ключевое слово ***mutable*** требует обеспечить такое хранение помеченного им поля данных, чтобы это поле можно было гарантированным образом модифицировать, даже для объектов, объявленных *константами*. Иными словами, ***mutable*** означает «никогда не может быть ***const***». В результате упрощается определение функции ***string_rep()***:

```
string Date::string_rep() const
{
  if(!cache_valid)
  {
    comute_cache_value();
    cache_valid = true;
  }
  return cache;
}
```

и становятся действительными все случаи ее разумного применения. Например:

```
Date d3;
const Date d4;
string s3 = d3.string_rep();
string s4 = d4.string_rep();             // ok!
```

Объявление части полей данных с модификатором ***mutable*** приемлемо в случаях, когда это малая часть общего числа полей данных класса. Если же у логически *константного* объекта большая часть полей подвержена изменениям, будет лучше все изменяющиеся поля переместить в отдельный объект с косвенным доступом. В этой технике наш пример с кэшируемой строкой принимает следующий вид:

```
struct cache
{
  bool valid;
  string rep;
};
```

```
class Date
{
  cache* c;                              // инициализируется конструктором (§10.4.6)
  void compute_cache_value () const;     // заполнить кэш
  // ...

public:
  // ...
  string string_rep () const;            // строковое представление
};

string Date::string_rep () const
{
  if(!c->valid)
  {
    compute_cache_value () ;
    c->valid = true;
  }
  return c->rep;
}
```

Различные обобщения техник кэширования приводят к тем или иным формам так называемых *ленивых вычислений* (*lazy evaluation*).

10.2.8. Структуры и классы

По определению *структура* есть класс с открытыми по умолчанию членами, так что

```
struct s { . . .
```

есть просто сокращенная форма записи для

```
class s {public: . . .
```

Спецификатор доступа ***private*** говорит, что все последующие за ним члены имеют закрытый режим доступа, точно так же, как ***public*** сообщает об открытом режиме доступа. За исключением различий в именах следующие объявления эквивалентны:

```
class Date1
{
  int d, m, y;

public:
  Date1 (int dd, int mm, int yy) ;
  void add_year (int n) ;              // прибавить n лет
};

struct Date2
{
private:
  int d, m, y;

public:
  Date2 (int dd, int mm, int yy) ;
  void add_year (int n) ;              // прибавить n лет
};
```

Какой стиль объявления использовать, зависит от вкуса и обстоятельств. Я обычно предпочитаю использовать *struct* для классов данных с открытыми членами. О таких типах я думаю как о «не совсем классах, просто наборах данных». Конструкторы и функции доступа полезны и в этих случаях, просто для удобства, а не как фундаментальные элементы типов (§24.3.7.1).

Вовсе не обязательно объявлять данные первыми. Часто полезно первыми поместить функции, чтобы подчеркнуть особую важность открытого интерфейса пользователя. Например:

```
class Date3
{
public:
  Date3 (int dd, int mm, int yy) ;
  void add_year (int n) ;            // прибавить n лет

private:
  int d, m, y;
};
```

В реальном коде, где и данные, и открытый интерфейс классов весьма велики по сравнению с учебным кодом, я предпочитаю стиль, использованный только что для объявления *Date3*.

Спецификаторы доступа могут встречаться в объявлениях классов много раз. Например:

```
class Date4
{
public:
  Date4 (int dd, int mm, int yy) ;

private:
  int d, m, y;

public:
  void add_year (int n) ;            // прибавить n лет
};
```

Множественные открытые и множественные закрытые секции класса (как в *Date4*) могут запутать программиста, но они нужны, например, для автоматической генерации кода.

10.2.9. Определение функций в теле определения класса

Функция-член, *определенная*, а не только объявленная в *теле определения класса*, по умолчанию *предполагается встраиваемой*. Ясно, что это полезно для небольших и часто применяемых функций. Как и определение класса, частью которого такие функции являются, они могут директивами *#include* включаться во многие единицы трансляции. Как и для класса в целом, их определение должно быть всюду одинаковым (§9.2.3).

Стиль, в котором поля данных класса помещаются в конец тела определения класса, вызывает небольшие проблемы в связи со встраиваемыми функциями-членами, ссылающимися на поля данных. Рассмотрим пример:

```
class Date
{
public:
  int day() const {return d; }          // возвращает Date::d
  // ...
private:
  int d, m, y;
};
```

Это абсолютно правильный код на C++, потому что функция-член класса может обращаться к любому члену класса так, как будто весь класс полностью определен до определения тела функции. Это, однако, может смутить человека, читающего программу.

Поэтому я либо помещаю поля данных в начало определения класса, либо располагаю определения встраиваемых функций-членов после определения класса:

```
class Date
{
public:
  int day() const;
  // ...

private:
  int d, m, y;
};

inline int Date::day() const {return d; }
```

10.3. Эффективные пользовательские типы

В предыдущем разделе отдельные части класса *Date* рассматривались в связи с изучением основных средств C++, предназначенных для определения классов. В настоящем разделе я переворачиваю акценты и рассматриваю в первую очередь простой и эффективный класс *Date* и показываю, как средства C++ поддерживают дизайн таких типов данных.

Небольшие, интенсивно используемые абстракции весьма типичны для большинства приложений: латинские буквы, китайские иероглифы, целые числа и числа с плавающей запятой, точки, указатели, координаты, преобразования, пары (указатель, смещение), даты, время, диапазоны, связи, ассоциации, узлы, пары (значение, единица измерения), местоположения на диске, расположение исходного кода, валюты, строки, прямоугольники, масштабируемые числа с фиксированной запятой, обыкновенные дроби, символьные строки, вектора и массивы. Каждое приложение использует хоть что-то из перечисленного набора. Часто некоторые из этих типов используются весьма интенсивно. Типичное приложение использует часть типов непосредственно, а часть — косвенно, через библиотеки.

Язык C++, как и другие языки программирования, непосредственно поддерживает лишь часть из перечисленных абстракций. Большинство же из подобного рода абстракций непосредственно не поддерживается ввиду их чрезвычайной многочисленности. Более того, разработчик универсального языка высокого уровня и не может в деталях представить себе все конкретные нужды каждого приложения. Вместо

этого, пользователю предоставляются стандартные механизмы для определения таких небольших, конкретных типов данных. Эти типы и принято называть конкретными типами или конкретными классами, чтобы противопоставить их абстрактным классам (§12.3) и классам из иерархий наследования (§12.2.4, §12.4).

При разработке языка С++ учитывалась необходимость предоставить средства определения и эффективного использования пользовательских типов данных. Это фундамент эффективного программирования. Ведь, как следует из статистики, простые и приземленные средства намного важнее сложных и изощренных.

В свете изложенного, давайте определим более совершенный вариант класса *Date*:

```
class Date
{
public:                                        // открытый интерфейс:
  enum Month {jan=1,feb,mar,apr,may,jun,ju,aug,sep,oct,nov,dec};
  class Bad_date { };                          // класс исключений
  Date(int dd =0, Month mm =Month(0), int yy =0); // 0 - "взять умолчат. значение"
  //Функции доступа к дате:
  int day() const;
  Month month() const;
  int year() const;
  string string_rep() const;                   // string представление
  void char_rep(char s[]) const;               // представление С-строкой
  static void set_default(int, Month, int);
  //Функции для изменения даты:
  Date& add_year(int n);                       // прибавить n лет
  Date& add_month(int n);                      // прибавить n лет
  Date& add_day(int n);                        // прибавить n лет
private:
  int d, m, y;                                 // представление
  static Date default_date;
};
```

Набор операций класса *Date* весьма типичен для любого пользовательского типа:

1. Конструктор, определяющий, как инициализируются объекты (переменные) типа.
2. Набор функций доступа. Эти функции снабжены модификатором *const*, который указывает, что функции не изменяют состояния объектов, для которых они вызваны.
3. Набор функций, позволяющий легко манипулировать датами без необходимости разбираться в деталях их устройства или осмысливать запутанную семантику типа.
4. Набор неявно определенных операций для свободного копирования объектов.
5. Класс *Bad_date*, используемый в исключениях с сообщениями об ошибках.

Я определил отдельный тип ***Month***, чтобы он имел дело с месяцами и помогал не путаться при записи, например, 7 июня, по американскому стилю в виде ***Date(6, 7)***, или по европейскому стилю в виде ***Date(7, 6)***.

Я также рассматривал возможность определения отдельных типов ***Day*** и ***Year***, чтобы избежать путаницы между вариантами ***Date(1995,jul,27)*** и ***Date(27,jul,1995)***. Однако эти типы не столь полезны, как ***Month***. Почти все ошибки такого рода выявляются в процессе выполнения — 26 июля 27 года не часто встречается в моей повседневной практике. Как обрабатывать даты до 1800 года или около того, вопрос довольно тонкий, и его лучше оставить историкам. Кроме всего, число месяца нельзя проверить в отрыве от самого месяца и даже года. В §11.7.1 приводится определение удобного в использовании типа ***Year***.

Где-то нужно задать корректную дату по умолчанию. Например:

```
Date Date::default_date(22,jan,1901);
```

Я здесь изъял технику кэширования, рассмотренную в §10.2.7.1, ибо в столь простом типе в ней нет необходимости. В то же время, ее всегда можно внедрить обратно как деталь реализации, не влияющую на открытый интерфейс пользователя.

Вот небольшой пример того, как тип ***Date*** может использоваться:

```
void f(Date& d)
{
  Date lvb_day = Date(16,Date::dec,d.year());
  if(d.day()==29 && d.month()==Date::feb)
  {
    //...
  }

  if(midnight()) d.add_day(1);
  cout<< "day after:" << d+1 << '\n';
}
```

Здесь предположено, что для типа ***Date*** определены операции << и +. Об этом рассказано в §10.3.3.

Обратите внимание на запись ***Date::feb***. Дело в том, что функция ***f()*** не является членом ***Date***, и поэтому в ней нужно явно указывать, что речь идет о ***feb*** из ***Date***, а не о какой-либо иной сущности.

Зачем нужно определять отдельный тип данных для такого простого понятия, как дата? Можно было бы ограничиться следующей структурой

```
struct Date
{
  int day, month, year;
};
```

и позволить программистам решать, что с ней делать. Однако если бы мы поступили таким образом, каждый пользователь должен был бы манипулировать полями ***Date*** непосредственно, или определить специальные функции для выполнения этих действий. В результате, понятие даты было бы «размазано» по всей системе; его было бы сложно понимать, документировать и модифицировать. Реализация концепции в виде простой структуры с неизбежностью приводит к дополнительной работе со стороны каждого пользователя структуры.

Несмотря на то, что класс ***Date*** и выглядит столь простым, все же нужно потрудиться, чтобы все работало как надо. Например, инкрементирование дат должно учитывать високосность года, что количество дней в месяцах разное и т.д. (§10.6[1]). Также, для многих приложений представление дат в виде «день-месяц-год» может оказаться недостаточным. Если мы решим изменить представление, то нам потребуется изменить лишь ограниченный набор функций. Например, чтобы представить ***Date*** в виде количества дней до или после 1 января 1970 года, мы должны будем изменить лишь функции-члены класса Date (§10.6[2]).

10.3.1. Функции-члены

Естественно, требуется выполнить и реализацию функций-членов класса ***Date***. Вот пример реализации конструктора класса ***Date***:

```
Date::Date(int dd, Month mm, int yy)
{
  if(yy == 0) yy = default_date.year();
  if(mm == 0) mm = default_date.month();
  if(dd == 0) dd = default_date.day();

  int max;

  switch(mm)
  {
    case feb:
      max = 28+leapyear(yy);
      break;
    case apr: case jun: case sep : case nov:
      max = 30;
      break;
    case jan: case mar: case may: case jul: case aug: case oct: case dec:
      max = 31;
      break;
    default:
      throw Bad_date();                    // кто-то шутит или напутал
  }

  if(dd<1 || max<dd) throw Bad_date();

  y = yy;
  m = mm;
  d = dd;
}
```

Конструктор проверяет, является ли полученная им дата допустимой. Если нет, как например в случае ***Date(30, Date::feb, 1994)***, он генерирует исключение (§8.3, глава 14), свидетельствующее о том, что что-то пошло не так и это лучше не игнорировать. Если же представленные на вход конструктору данные корректны, то производится очевидная инициализация. В целом, инициализация получилась относительно сложной как раз из-за проверки допустимости даты. Это достаточно типичный случай. С другой стороны, после того, как дата создана, ее можно использовать и копировать без дополнительных проверок. Другими словами, конструктор устанавливает инвариант класса (в данном случае это корректность даты). Остальные функ-

ции-члены могут полагаться на этот инвариант и должны поддерживать его. Такая техника проектирования упрощает код в заметной степени (см. §24.3.7.1).

Для выбора месяца по умолчанию я использую выражение ***Month***(***0***), не представляющее реального месяца. Я мог бы специально для этой цели предусмотреть элемент в перечислении ***Month***. Но я решил, что лучше уж использовать явно аномальное значение, чем представить дело так, что в году 13 месяцев. Обратите внимание, что нулем можно пользоваться потому, что он гарантированно находится внутри диапазона значений перечисления ***Month*** (§4.8).

Я думал над тем, чтобы вынести проверку корректности дат в отдельную функцию ***is_date***(). Но затем я понял, что при этом пользовательский код станет более сложным и менее надежным, чем в случае опоры на исключения, как в следующем примере, предполагающем, что операция >> определена для типа ***Date***:

```
void fill(vector<Date>& a)
{
 while(cin)
 {
   Date d;
   try
   {
     cin >> d;
   }
   catch(Date::Bad_date)          // обработка ошибки
   {
     continue;
   }
   aa.push_back(d);               // см. §3.7.3
 }
}
```

Что вообще типично для простых конкретных типов, определения функций-членов характеризуются диапазоном оценок от «тривиально» до «не слишком сложно». Например:

```
inline int Date::day() const
{
 return d;
}

Date& Date::add_month(int n)
{
 if(n==0) return *this;

 if(n>0)
 {
   int delta_y = n/12;
   int mm = m+n%12 ;
   if(12 < mm)                    // обратите внимание: int(dec)==12
   {
     delta_y++ ;
     mm -= 12;
   }
```

```
        // работа со случаями, когда день d не существует для Month(mm):
        y += delta_y;
        m = Month (mm) ;
        return *this;
    }

    // отрицательное n:
    return *this;
}
```

10.3.2. Функции поддержки (helper functions)

В типичном случае с классом логически связан набор вспомогательных *функций поддержки* (*helper functions*), определять которые внутри класса нет необходимости, так как они не нуждаются в прямом доступе к внутреннему представлению класса. Например:

```
int diff(Date a, Date b) ;         // количество дней между a и b
bool leapyear (int y) ;
Date next_weekday (Date d) ;
Date next_saturday (Date d) ;
```

Определение таких функций в качестве функций-членов лишь усложнило бы интерфейс класса и потребовало бы большей работы по просмотру всех функций, подлежащих модификации при изменении внутреннего представления класса.

Но как такие функции следует «связывать» с классом? Традиционно, их объявления просто помещают в тот же самый файл, где объявляется сам класс, так что пользователям они становятся доступными с помощью той же самой директивы **#include**, с помощью которой они включают определение интерфейса класса (§9.2.1). Например:

```
#include "Date.h"
```

Вместо использования заголовочного файла **Date.h** — или в качестве альтернативы — мы могли бы установить связь явно, поместив объявления функций поддержки в одно пространство имен с объявлением класса (§8.2):

```
namespace Chrono                  // средства для работы со временем
{
    class Date { /* ... */ };

    int diff(Date a, Date b) ;
    bool leapyear (int y) ;
    Date next_weekday (Date d) ;
    Date next_saturday (Date d) ;
    // ...
}
```

В типичных случаях пространство имен **Chrono** содержало бы еще и другие логически связанные классы, например, **Time** и **Stopwatch**, а также их собственные функции поддержки. Применение пространства имен лишь для одного класса является избыточным и ведет к напрасному переусложнению.

10.3.3. Перегруженные операции

Полезно добавить к классу функции, обеспечивающие возможность пользовательскому коду применять привычные формы записи. Например, функция ***operator==()*** определяет, как *операция* == (операция сравнения на равенство) работает для объектов класса ***Date***:

```
inline bool operator== (Date a, Date b)      // проверка на равенство
{
  return a.day()==b.day() && a.month()==b.month() && a.year()==b.year();
}
```

Другими очевидными кандидатами являются:

```
bool operator!= (Date, Date);                // не равно
bool operator< (Date, Date);                 // меньше чем
bool operator> (Date, Date);                 // больше чем
// ...
Date& operator++ (Date& d);                  // увеличить Date на один день
Date& operator-- (Date& d);                  // уменьшить Date на один день

Date& operator+= (Date& d, int n);           // прибавить n дней
Date& operator-= (Date& d, int n);           // вычесть n дней

Date operator+ (Date d, int n);              // прибавить n дней
Date operator- (Date d, int n);              // вычесть n дней

ostream& operator<< (ostream&, Date d);      // вывести d
istream& operator>> (istream&, Date& d);     // считать d
```

Для класса ***Date*** перегрузка этих операций совершается исключительно ради дополнительного удобства. Но для таких типов, как комплексные числа (§11.3), вектора (§3.7.1) и классы функторов (функциональных объектов или объектов-функций) (§18.4) — стандартные операции настолько проникли в сознание, что их перегрузка является просто обязательной. Перегрузка операций подробно рассматривается в главе 11.

10.3.4. Роль конкретных классов

Я называю простые типы, определяемые пользователем, такие как ***Date***, *конкретными типами* (*concrete types*), чтобы противопоставить их *абстрактным классам* (*abstract classes*) (§2.5.4) и классовым иерархиям (§12.3), а также подчеркнуть их сходство со встроенными типами, такими как ***int*** или ***char***. Конкретные типы называют также *типами-значениями* (*value types*), а их применение в программе — программированием, ориентированным на значения (value-oriented programming). Модель их применения и философия, стоящая за проектированием таких типов, сильно отличаются от того, что принято называть объектно-ориентированным программированием (§2.6.2).

Конкретные типы предназначены для того, чтобы хорошо и эффективно выполнять отдельную небольшую работу. Как правило, пользователю не предоставляют возможностей для модификации поведения конкретных типов. В частности, конкретные типы не предназначаются для демонстрации полиморфного поведения (см. §2.5.5, §12.2.6).

Если вам не подходят отдельные черты конкретного типа, вы строите новый тип с желаемым поведением. При этом вы можете повторно применить старый конкретный тип (reuse a concrete type) в качестве строительного кирпичика нового типа точно так же, как и встроенный тип, такой как *int*. Например:

```
class Date_and_time
{
private:
  Date d;
  Time t;

public:
  Date_and_time(Date d, Time t);
  Date_and_time(int d, Date::Month m, int y, Time t);
  //...
};
```

Механизм наследования классов, обсуждаемый в главе 12, может быть применен с целью формулирования различий (и сходства) нового типа и заданного конкретного типа. Определение типа *Vec* из типа *vector* (§3.7.2) служит соответствующим примером.

При наличии качественного компилятора конкретные типы вроде *Date* не вносят дополнительных накладных расходов ни в плане скорости выполнения, ни в плане объема занимаемой памяти. Размер конкретного типа известен во время компиляции, так что объекты этих типов можно создавать в стеке (то есть без операций размещения в свободной памяти). Раскладка объектов этих типов известна во время компиляции, так что не составляет труда реализовывать встраиваемые операции. Аналогично, совместимость с языками типа C или Fortran достигается без дополнительных усилий.

Хороший набор конкретных типов может составить надежный фундамент приложения. Отсутствие маленьких эффективных типов ведет к перерасходу компьютерной памяти и вычислительных мощностей из-за применения слишком общих и громоздких классов. Кроме того, отсутствие конкретных типов приводит к снижению эффективности труда программиста, поскольку каждый программист вынужден в очередной раз писать свой собственный код для манипулирования «простыми и широко применяемыми» структурами данных.

10.4. Объекты

Объекты можно создавать разными способами. Некоторые объекты являются локальными переменными, некоторые — глобальными переменными, другие входят в состав более крупных объектов иных классов (являются членами классов) и т.д. В настоящем разделе обсуждаются перечисленные альтернативы, правила, которыми при этом руководствуются, как конструкторы используются для инициализации объектов, и как деструкторы очищают выделенные под объекты ресурсы перед тем, как объекты станут недоступными.

10.4.1. Деструкторы

Конструкторы предназначены для инициализации объектов. Можно сказать, они создают среду, в которой затем выполняются функции-члены. Иногда создание такой среды включает выделение таких ресурсов, как файлы, блокировки или память, и которые должны быть освобождены после использования объектов класса (§14.4.7). Таким образом, эти классы нуждаются в функции, которая гарантированно вызывается при уничтожении объектов аналогично тому, как конструктор гарантированно вызывается при их создании. Такие функции принято называть *деструкторами* (*destructors*). Их типичное поведение заключается в очистке и/или освобождении ресурсов. Деструкторы автоматически вызываются, когда локальные переменные выходят из области видимости, или когда явным образом уничтожаются объекты, динамически созданные в свободной памяти. Только в крайне необычных ситуациях от пользователя требуется вызывать деструкторы явно (§10.4.11).

Чаще всего деструкторы освобождают память, динамически выделенную конструкторами. Рассмотрим простую таблицу элементов некоторого типа ***Name***. Конструктор класса ***Table*** должен выделить память под элементы таблицы. Когда таблицу тем или иным способом уничтожают, мы должны быть уверены в том, что ранее выделенная в конструкторе память возвращена системе с целью ее дальнейшего повторного использования. Это можно сделать, предоставив специальную, комплиментарную к конструктору функцию:

```
class Name
{
  const char* s;
  // ...
};

class Table
{
  Name* p;
  size_t sz;
public:
  Table (size_t s = 15) {p = new Name[sz = s] ; }    // конструктор
  ~Table () {delet[ ] p; }                            // деструктор

  Name* lookup (const char* ) ;
  bool insert (Name* ) ;
};
```

Тильда в имени деструктора ***~Table***() является знаком операции дополнения, что намекает на характер этой функции, дополняющей классовый конструктор ***Table***().

Комплиментарные пары конструктор/деструктор служат обычным для языка C++ механизмом реализации объектов переменного размера. Контейнеры стандартной библиотеки, такие как ***map***, применяют некоторые вариации этой техники для управления выделением памяти под элементы, так что последующее обсуждение иллюстрирует технику, которой вы фактически пользуетесь всегда, когда используете стандартные контейнеры (и стандартный тип ***string*** в том числе). Это обсуждение в равной степени распространяется и на типы без явно запрограммиро-

ванных деструкторов, ибо в этом случае можно предполагать, что деструктор просто ничего не делает.

10.4.2. Конструкторы по умолчанию

Аналогично, можно считать, что большинство типов имеют конструктор по умолчанию. Конструктор по умолчанию можно вызывать, не предоставляя никаких аргументов. В вышеприведенном примере из-за наличия умолчательного значения 15 для аргумента конструктора *Table::Table(size_t)*, его можно считать конструктором по умолчанию. Если пользователь явным образом определяет в классе конструктор по умолчанию, именно этот вариант и задействуется; в противном случае компилятор пытается сгенерирвать свой вариант умолчательного конструктора, но только в случаях, когда отсутствуют явно определенные в классе конструкторы иных типов. Сгенерированный компилятором конструктор по умолчанию пытается неявно вызывать умолчательные конструкторы для членов класса (также имеющими тип классов) и конструкторы базовых классов (§12.2.2). Например:

```
struct Tables
{
  int i;
  int vi[10];
  Table t1;
  Table vt[10];
};

Tables tt;
```

Здесь *tt* будет инициализироваться сгенерированным конструктором по умолчанию, который вызовет *Table(15)* для *tt.t1* и каждого элемента *tt.vt*. С другой стороны, *tt.i* и элементы *tt.vi* не относятся к классовым типам и, соответственно, не инициализируются. Причины такого расхождения заключаются в необходимости обратной совместимости с языком С и опасении излишних накладных расходов для встроенных типов данных.

Из-за того, что *константы* и ссылки обязаны инициализироваться (§5.5, §5.4), класс, содержащий члены, являющиеся константами или ссылками, не может конструироваться по умолчанию, если только программист не предоставит явным образом соответствующий конструктор (§10.4.6.1). Например:

```
struct X
{
  const int a;
  const int& r;
};

X x;            // error: нет умолчательного конструктора для X
```

Конструкторы по умолчанию можно вызвать явно (§10.4.10). Встроенные типы также обладают конструкторами по умолчанию (§6.2.8).

10.4.3. Конструирование и уничтожение объектов

Рассмотрим различные способы создания и последующего уничтожения объектов классов. Объект может быть создан в качестве:

- §10.4.4. Именованного автоматического объекта, создаваемого каждый раз при проходе потока управления работающей программы через его определение, и уничтожаемого по выходу из блока, содержащего указанное определение.
- §10.4.5. Объекта в свободной памяти, создаваемого операцией ***new*** и уничтожаемого операцией ***delete***.
- §10.4.6. Нестатического члена классового типа, создаваемого в виде подобъекта в момент создания объемлющего объекта иного типа, и уничтожаемого в момент уничтожения указанного объемлющего объекта.
- §10.4.7. Элемента массива, создаваемого и уничтожаемого в моменты создания и уничтожения самого массива.
- §10.4.8. Локального статического объекта, создаваемого в момент первого прохождения потока управления работающей программы через его определение, и уничтожаемого в момент окончания работы программы.
- §10.4.9. Глобального объекта, объекта из пространства имен или статического объекта класса, создаваемого при старте программы и уничтожаемого при завершении работы программы.
- §10.4.10. Временного объекта, создаваемого в процессе вычисления выражения, и уничтожаемого после окончания вычисления полного выражения.
- §10.4.11. Объекта, помещаемого в область памяти, предоставляемую функцией пользователя, с учетом параметров, передаваемых конструктору операцией размещения (placement new).
- §10.4.12. Члена объединения (***union***), который не может иметь ни конструктора, ни деструктора.

Этот список приблизительно отсортирован в порядке важности. В следующих подразделах подробно рассматриваются все перечисленные способы создания объектов и их использование.

10.4.4. Локальные объекты

Конструктор локального объекта выполняется каждый раз, когда поток управления работающей программы проходит через его определение. Деструктор же вызывается при каждом выходе из блока, содержащего указанное объявление. Деструкторы локальных объектов выполняются в порядке, обратном выполнению их конструкторов. Например:

```
void f(int i)
{
  Table aa;
  Table bb;
  if(i>0)
  {
    Table cc;
```

```
    // ...
  }
  Table dd;
  // ...
}
```

Здесь ***aa***, ***bb*** и ***dd*** конструируются (в указанном порядке) каждый раз при вызове функции ***f***(), и ***dd***, ***bb*** и aa уничтожаются (в указанном порядке) при каждом выходе из ***f***(). Если же в конкретном вызове ***i>0***, сс конструируется после bb, и уничтожается перед конструированием ***dd***.

10.4.4.1. Копирование объектов

Если ***t1*** и ***t2*** являются объектами класса ***Table***, то присваивание ***t2=t1*** по умолчанию означает почленное (побитовое) копирование ***t1*** в ***t2*** (§10.2.5). Такое поведение операции присваивания может вызывать проблемы для объектов классов, содержащих указатели в качестве членов. Почленное копирование не соответствует семантике копирования объектов классов, управляющих ресурсами с помощью пары конструктор/деструктор. Например:

```
void h ()
{
  Table t1;
  Table t2 = t1;                  // копирующая инициализация: проблема

  Table t3;
  t3 = t2;                        // копирующее присваивание: проблема
}
```

Умолчательный конструктор класса ***Table*** вызывается дважды — при конструировании объектов ***t1*** и ***t3***; для объекта ***t2*** он не вызывается, так как тут работает копирование (из ***t1*** в ***t2***). В то же время деструктор класса ***Table*** вызывается трижды — при уничтожении объектов ***t1***, ***t2*** и ***t3***! Согласно умолчательной трактовке операции присваивания перед выходом из функции ***h***() каждый из объектов ***t1***, ***t2*** и ***t3*** будет содержать указатель на массив имен, память под который была динамически выделена в свободной памяти в момент создания ***t1***. Указатель же на массив имен, выделенный в свободной памяти при создании ***t3***, не сохранился из-за того, что была выполнена операция ***t3=t2*** с ее почленным копированием. Ввиду отсутствия *автоматической сборки мусора* (*garbage collection*; §10.4.5), эта память *потеряна для программы* навсегда. С другой стороны, массив, выделенный для ***t1***, теперь адресуется также из ***t2***, ***t3***, и в итоге он будет удаляться трижды. Результирующий эффект стандартом не определен, но почти наверняка будет катастрофическим.

Подобного рода аномалий можно избежать, если явно определить, что же нужно понимать под присваиванием объектов класса ***Table***:

```
class Table
{
  // ...как в §10.4/1...
  Table (const Table&) ;                  // копирующий конструктор
  Table& operator= (const Table&) ;       // присваивание
};
```

Программист волен определить любое приемлемое поведение для операции присваивания, однако традиционным является копирование содержимого контейнера (или создание иллюзии у пользователя о таком копировании; см. §11.12). Например:

```
Table::Table(const Table& t)              // копирующий конструктор
{
 p = new Name[sz=t.sz];
 for(int i=0; i<sz; i++) p[i]=t.p[i];
}

Table& Table::operator=(const Table& t)   // присваивание
{
 if(this != &t)   // не забудьте о возможности самоприсваивания: t = t
 {
   delet[] p;
   p = new Name[sz=t.sz];
   for(int i=0; i<sz; i++) p[i]=t.p[i];
 }
 return*this;
}
```

Копирующий конструктор (*copy constructor*) и *операция присваивания* (*assignment*) различаются в следующем принципиальном моменте: копирующий конструктор инициализирует *«сырую»* (*raw — ранее не инициализированную*) *память*, в то время как операция присваивания работает над участком памяти, содержащем корректно сконструированный классовый объект.

В ряде случаев допускаются разные схемы оптимизации, но основная стратегия работы операции присваивания проста: *защита от присваивания самому себе*, удаление старых элементов, инициализация и копирование новых элементов. Как правило, нужно скопировать все нестатические члены (§10.4.6.3). Для сообщения об ошибках копирования можно использовать исключения (§14.4.6.2). По поводу техники написания операций копирования, оставляющих левый операнд в корректном состоянии при возникновении исключений (exception-safe copy operations), см. §E.3.3.

10.4.5. Динамическое создание объектов в свободной памяти

Конструктор для объекта, создаваемого в свободной памяти, вызывается по операции ***new***. Объект в свободной памяти существует до тех пор, пока не будет вызвана операция ***delete*** с операндом, являющимся указателем на этот объект. Рассмотрим следующий пример:

```
int main()
{
  Table* p = new Table;
  Table* q = new Table;

  delete p;
  delete p;   // вероятно, приведет к ошибке во время выполнения
}
```

Конструктор ***Table*::*Table*()** вызывается дважды, как и деструктор ***Table*::~*Table*()**. К сожалению, операции ***new*** и ***delete*** не точно соответствуют друг другу в данном примере: объект, адресуемый указателем ***p***, удаляется дважды, а объект, адресуемый указателем ***q*** — ни разу. То, что объект не удаляется операцией ***delete***, не является ошибкой с формальной точки зрения языка C++; это лишь напрасный перерасход памяти (образно говорят об *утечке памяти* — *memory leak*). Но для программы, предлагаемой к длительному непрерывному использованию, это сильно вредит производительности и может трактоваться как откровенный брак в ее реализации. Существуют средства для помощи в нахождении ошибок, приводящих к утечкам памяти. Двойное освобождение памяти является серьезной операционной ошибкой с непредсказуемыми результатами, чаще всего катастрофическими.

Некоторые реализации C++ могут предоставлять услуги автоматической сборки мусора, сканируя объекты в свободной памяти с целью нахождения неиспользуемых объектов. Поведение сборщиков мусора не стандартизовано. При наличии автоматически работающих сборщиков мусора операция ***delete*** может спровоцировать ошибку двойного удаления объектов. Чаще всего это можно рассматривать как мелкое неудобство, ибо если известно, что работает сборщик мусора, деструкторы, освобождающие память, могут быть просто опущены. Конечно, все это ведет к непереносимости программ, к некоторой потере производительности и, возможно, к неточно предсказуемому их поведению (§C.9.1).

После того как выполнена операция ***delete***, к объекту нельзя обращаться никоим образом. К сожалению, реализации не в состоянии надежно выявлять такого рода логические ошибки на стадии компиляции.

Пользователь может явным образом определять поведение операций ***new*** и ***delete*** (см. §6.2.6.2 и §15.6). Также возможно определять варианты взаимодействия операций выделения памяти, конструирования (инициализации) и генерации/обработки исключений (см. §14.4.5 и §19.4.5). Создание массивов в свободной памяти обсуждается в §10.4.7.

10.4.6. Классовые объекты как члены классов

Рассмотрим класс, который содержит информацию о небольшой организации:

```
class Club
{
  string name;
  Table members;
  Table officers;
  Date founded;
  // ...
  Club(const string& n, Date fd);
};
```

Конструктор класса ***Club*** принимает в качестве аргументов имя клуба и дату его основания. Аргументы конструкторам членов класса ***Club*** передаются с помощью *списка инициализации членов* (*member initializer list*) в определении конструктора объемлющего их класса:

```
Club::Club(const strin& n, Date fd)
    : name(n), members(), officers(), founded(fd)
{
  //...
}
```

Инициализаторы членов класса *предваряются двоеточием* и отделяются друг от друга запятыми.

Конструкторы членов класса вызываются до момента выполнения тела конструктора содержащего их класса. Порядок выполнения конструкторов членов соответствует порядку их объявления в определении объемлющего класса, а не порядку появления инициализаторов в списке инициализации. Во избежание путаницы, лучше упорядочить список инициализации в точном соответствии с порядком объявления членов. Порядок вызова деструкторов членов обратный по отношению к вызову их конструкторов, и выполняются они после выполнения тела деструктора объемлющего класса.

Если конструктор члена класса не требует аргументов, то в списке инициализации соответствующий ему инициализатор можно просто опустить:

```
Club::Club(const string& n, Date fd) : name(n), founded(fd)
{
  //...
}
```

Данная версия конструктора полностью эквивалентна его предыдущей версии. В обеих версиях член officers инициализируется умолчательным конструктором класса ***Table*** (для этого конструктора предусмотрено умолчательное значение аргумента, равное 15).

Когда уничтожается объект объемлющего класса (например, класса ***Club***), сначала выполняется тело деструктора этого класса (если таковой определен), а затем выполняются деструкторы членов в порядке, обратном порядку вызова их конструкторов (то есть в порядке, обратном объявлению членов в объемлющем классе). Можно сказать, что конструктор собирает среду исполнения функций-членов в порядке снизу-вверх (сначала конструкторы членов, а затем конструктор объемлющего класса), в то время как деструктор демонтирует эту среду сверху вниз.

10.4.6.1. Обязательная инициализация членов

Инициализаторы членов необходимы (обязательны) в тех случаях, когда инициализация отличается от присваивания — для членов класса с типами, не имеющими умолчательных конструкторов, для ***константных*** членов и для ссылок. Например:

```
class X
{
  const int i;
  Club cl;
  Club& rc;
  //...
  X(int ii, const string& n, Date d, Club& c) : i(ii),cl(n,d),rc(c) {}
};
```

Не существует никаких других способов инициализации таких членов, а отсутствие их инициализации является ошибкой. Однако для большинства иных типов существует выбор между применением списка инициализации и присваиваниями. В таких случаях я предпочитаю использовать список инициализации, делая тем самым факт инициализации особо наглядным. Часто это приводит и к более эффективному коду. Например:

```
class Person
{
  string name;
  string address;
  // ...
  Person(const Person&);
  Person(const string& n, const string& a);
};

Person::Person(const string& n, const string& a)
      : name(n)
{
  address = a;
}
```

Здесь ***name*** инициализируется копией ***n***. А член ***address*** сначала инициализируется пустой строкой, а затем ему присваивается значение ***a***.

10.4.6.2. Члены-константы

Статические члены-константы интегральных типов можно инициализировать с помощью константных выражений *прямо в их объявлениях*. Например:

```
class Curious
{
public:
  static const int c1 = 7;        // ok, но нужно помнить определение
  static int c2 = 11;             // error: не константа
  const int c3 = 13;              // error: нет модификатора static
  static const int c4=f(17);      // error: не константный инициализатор
  static const float c5 = 7.0;    // error: не интегральный тип
  // ...
};
```

Если (и только если) требуется инициализированный член хранить и использовать как объект в памяти, то его где-либо нужно уникально определить. Инициализатор при этом повторять нельзя:

```
const int Curious::c1;            // обязательно (но без повтора инициализатора)
const int* p = &Curious::c1;      // ok: Curious::c1 был определен
```

В качестве альтернативы, в классе можно определить символические константы с помощью перечислений (§4.8, §14.4.6, §15.3). Например:

```
class X
{
  enum {c1 = 7, c2 = 11, c3 = 13, c4 = 17};
  //...
};
```

Тогда у вас не будет возникать искушение инициализировать в классе переменные, числа с плавающей запятой и т.д.

10.4.6.3. Копирование членов

Умолчательный копирующий конструктор и умолчательный вариант операции присваивания (§10.4.4.1) просто копируют все элементы класса. Когда такое копирование не проходит, при попытке копирования объектов класса возникает ошибка. Например:

```
class Unique_handle
{
private:                // копирующие операции здесь private для предотвращения явного
                        // копирования (§11.2.2)

  Unique_handle(const Unique_handle&) ;
  Unique_handle& operator=(const Unique_handle&) ;

public:
  // ...
};

struct Y
{
  // ...
  Unique_handle a;   // требуется явная инициализация
};

Y y1;
Y y2 = y1;              // error: нельзя скопировать Y::a
```

Кроме того, умолчательный вариант присваивания не проходит в тех случаях, когда нестатические члены являются ссылками, константами или имеют пользовательский тип, не определяющий умолчательного варианта операции присваивания.

Обратите внимание, что в результате работы умолчательного копирующего конструктора члены-ссылки в обоих объектах (оригинальном и копии) ссылаются на один и тот же объект в памяти, что порождает проблемы при уничтожении этого объекта.

При написании копирующего конструктора нужно проследить за тем, чтобы были скопированы все элементы, для которых действительно требуется копирование. Иначе они проинициализируются умолчательными значениями, а это часто не то, что требуется. Например:

```
Person::Person(const Person& a) : name(a.name) { }     // будьте внимательны!
```

Здесь я забыл скопировать ***address***, и по умолчанию ***address*** проинициализируется пустой строкой. При добавлении нового члена к классу всегда тщательно проверяйте, не нужно ли при этом модифицировать ранее определенные в классе конструкторы для инициализации и копирования нового члена.

10.4.7. Массивы

Если объект класса можно создать без указания явного инициализатора, то можно определить массив элементов этого класса. Например:

```
Table tbl[10];
```

Создается массив из **10 элементов** типа **Table**, и каждый элемент инициализируется вызовом **Table::Table()** с умолчательным аргументом 15.

За исключением применения списка инициализации (§5.2.1, §18.6.7) не существует иных способов явного задания аргументов конструктора в объявлении массива. Если в обязательном порядке требуется инициализировать элементы массива разными значениями, нужно написать соответствующий умолчательный конструктор, который так или иначе вырабатывает необходимые значения. Например:

```
class Ibuffer
{
  string buf;

public:
  Ibuffer() {cin>>buf; }
  //...
};

void f()
{
  Ibuffer words[100];    // каждое слово инициализируется из cin
  //...
}
```

Таких хитростей, все же, лучше избегать.

Когда массив уничтожается, деструктор вызывается для каждого элемента массива. Это происходит неявным образом для массивов, память под которые не выделялась операцией **new**. Как и язык C, язык C++ не отличает указатель на массив от указателя на первый элемент массива (§5.3). Как следствие, программист должен явно указать, что удаляется — массив или отдельный элемент. Например:

```
void f(int sz)
{
  Table* tl = new Table;
  Table* t2 = new Table[sz];
  Table* t3 = new Table;
  Table* t4 = new Table[sz};

  delete tl;              // правильно
  delete[] t2;            // правильно
  delete[] t3;            // неправильно
  delete t4;              // неправильно
}
```

Точные детали выделения памяти под массивы или под отдельные объекты зависят от конкретной реализации. Поэтому и реакция на ошибочное применение операций **delete** или **delete[]** будет также различаться. В простейших и неинтересных случаях типа приведенного выше примера компилятор в состоянии обнару-

жить потенциальные проблемы, но в общем случае нечто ужасное произойдет лишь во время выполнения программы.

С логической точки зрения нет необходимости в особой форме операции ***delete*** `[]`, предназначенной специально для массивов. С другой стороны, представим, что мы потребовали от системы управления свободной памятью маркировать выделяемые блоки памяти таким образом, чтобы сразу было ясно, происходило это выделение для массивов или нет. С программиста при этом будет снята дополнительная ноша, однако она тяжелым грузом ляжет на приложения и снизит эффективность их выполнения.

Если же вы находите массивы в С-стиле громоздкими и неуклюжими, используйте вместо них некоторый подходящий класс, например ***vector*** (§3.7.1, §16.3):

```
void g ()
{
  vector<Table> v (10) ;                        // нет необходимости в удалении
  vector<Table>* p = new vector<Table> (10) ;   // используйте delete, а не delete[]

  delete p;
}
```

Применять контейнеры типа vector намного проще, чем то и дело манипулировать операциями ***new/delete***. Кроме того, класс ***vector*** обеспечивает безопасное поведение в плане исключений (приложение Е).

10.4.8. Локальные статические объекты

Конструктор для локального статического объекта (§7.1.2) вызывается только тогда, когда поток управления в работающей программе первый раз проходит через определение объекта. Рассмотрим пример:

```
void f(int i)
{
  static Table tbl;
  //...
  if(i)
  {
    static Table tbl2;
    //...
  }
}

int main ()
{
  f(0) ;
  f(1) ;
  f(2) ;
  //...
}
```

Здесь конструктор для ***tbl*** вызывается только при первом вызове функции ***f***`()`. Так как переменная ***tbl*** объявлена статической, она не уничтожается при выходе из ***f***`()`, и она повторно не конструируется при повторных вызовах ***f***`()`. Поскольку блок кода, содержащий определение ***tbl2***, не выполняется при вызове ***f***`(`***0***`)`, то ***tbl2*** не соз-

дается до вызова *f(1)*. При повторных входах в этот блок, имеющих место при последующих вызовах *f()*, ***tbl2*** повторно не конструируется.

При завершении работы программы деструкторы локальных статических объектов вызываются в порядке, обратном порядку их конструирования (§9.4.1.1). Однако точный момент их вызова не специфицируется.

10.4.9. Нелокальные объекты

Переменная, определяемая вне функций (в глобальном пространстве, в пространстве имен или классовая ***статическая*** переменная; §C.9), инициализируется (конструируется) до вызова функции ***main***(), а их деструкторы вызываются после завершения работы функции ***main***(). Динамическая компоновка программных модулей слегка усложняет картину, откладывая инициализацию до того момента, когда код будет динамически скомпонован с исполняемой программой.

Конструкторы нелокальных объектов в рамках единицы трансляции исполняются в порядке, в каком появляются их определения. Рассмотрим пример:

```
class X
{
  // ...
  static Table memtbl;
};

Table tbl;
Table X::memtbl;

namespace Z
{
  Table tbl2;
}
```

Здесь порядок создания объектов следующий: сначала ***tbl***, затем ***X::memtbl***, а затем ***Z::tbl2***. Заметьте, что объявления (в противоположность определениям), например объявление ***memtbl*** в ***X***, не влияют на порядок создания объектов. Деструкторы объектов вызываются в обратном порядке: сначала для объекта ***Z::tbl2***, затем для ***X::memtbl***, и потом для ***tbl***.

Нет никаких правил для порядка конструирования нелокальных объектов, определенных в разных единицах трансляции. Например:

```
// файл file1.c:
Table tbl1;

// файл file2.c:
Table tbl2;
```

Что создается раньше, ***tbl1*** или ***tbl2***, зависит от конкретной реализации. Даже в рамках одной и той же реализации нельзя точно специфицировать порядок их создания. Динамическая компоновка, или даже небольшие вариации в процессе компиляции способны изменить этот порядок. Аналогично, порядок уничтожения объектов также зависит от реализации.

При создании библиотеки часто бывает необходимо или просто удобно ввести тип, единственной целью которого является выполнение инициализации и завершающей очистки (cleanup). Такой тип предполагается использовать один раз при

создании статического объекта, чтобы обеспечить вызовы конструктора и деструктора. Вот пример на эту тему:

```
class Zlib_init
{
  Zlib_init() ;          // готовит Zlib к использованию
  ~Zlib_init() ;         // очищает ресурсы после использования Zlib
};

class Zlib
{
  static Zlib_init x;
  // ...
};
```

К сожалению, нет гарантии, что этот статический объект будет проинициализирован до его первого использования и уничтожен после последнего, если программа состоит из нескольких раздельно компилируемых единиц. Конкретная реализация может предоставить такую гарантию, но в общем случае такой гарантии нет. Программист можем сам обеспечить такую гарантию с помощью стратегии, обычно применяемой к локальным статическим объектам: стратегии флага первого использования. Например:

```
class Zlib
{
  static bool initialized;
  static void initialize() {initialized = true; }

public:
  // конструктор отсутствует
  void f()
  {
    if(initialized == false) initialize() ;
    // ...
  }
  // ...
};
```

Если многие функции потребуют проверки флага первого использования, то стратегия станет утомительной, но все же по-прежнему работоспособной. Она опирается на тот факт, что статические объекты без конструкторов инициализируются нулевыми значениями. Большие сложности начинаются тогда, когда первый вызов критичен по времени выполнения, и дополнительные накладные расходы на проверку и возможную инициализацию становятся обременительными. В таких случаях помогут дополнительные трюки (§21.5.2).

Альтернативным подходом в случае простых объектов является их представление в виде функций (§9.4.1):

```
int& obj() {static int x = 0; return x; }    // инициализация по первому вызову
```

Флаги первого использования не решают абсолютно всех проблем. Например, речь может пойти об объектах, которые ссылаются друг на друга во время конструирования. Конечно, таких ситуаций лучше избегать, но если все же они необходимы,

то конструирование нужно выполнять аккуратно и поэтапно. Еще стоит отметить, что нет аналогичной простой конструкции с флагом последнего использования (см. §9.4.1.1 и §21.5.2).

10.4.10. Временные объекты

Временные объекты (*temporary objects*) чаще всего создаются в процессе вычисления арифметических выражений. Например, при вычислении выражения *x*y+z* надо где-то сохранить промежуточный результат умножения *x*y*. За исключением случаев, когда эффективность выполнения кода особо критична (§11.6), временные объекты не беспокоят программиста. Однако ж, иногда на них приходится обращать внимание (§11.6, §22.4.7).

Если временный объект не связан ссылкой и не используется для инициализации именованного объекта, он обычно уничтожается по достижении конца полного выражения, в рамках оценки которого он был создан. *Полное выражение* (*full expression*) — это выражение, которое не является подвыражением (частью) другого выражения.

Стандартный класс *string* имеет функцию-член *c_str*(), которая возвращает массив символов в С-стиле, то есть с терминальным нулем (§3.5.1, §20.4.1). В нем также определена операция +, означающая конкатенацию строк. Все это важные и полезные черты класса *string*. Однако их совместное использование порождает малоизвестную проблему. Вот пример:

```
void f(string& s1, string& s2, string& s3)
{
  const char* cs = (s1+s2).c_str();
  cout<< cs;

  if( strlen(cs=(s2+s3).c_str())<8 && cs[0]=='a' )
  {
    // cs используется здесь
  }
}
```

Вероятно вашей первой реакцией будет «не надо так писать», и я абсолютно согласен с вашей реакцией. Но все же стоит рассмотреть этот пример подробнее.

Создается временный объект класса *string* для хранения результата конкатенации *s1+s2*. Потом указатель на С-строку извлекается из этого объекта. Затем — в конце выражения — временный объект уничтожается. Теперь зададимся вопросом, где же хранилась возвращаемая с помощью *c_str*() строка С-стиля? Она, вероятно, хранилась как часть временного объекта, содержащего результат конкатенации *s1+s2*. Но этот объект уничтожен, так что cs указывает на освобожденную память с непредсказуемым содержимым. Возможно, операция *cout<<cs* и сработает как надо, но это будет в любом случае чистой удачей. Компилятор может обнаруживать и предупреждать о большинстве подобного рода проблем.

Пример с оператором *if* тоньше. Само условие отработает как надо, ибо полным выражением, содержащим временный объект, хранящий результат конкатенации *s2+s3*, является само условие. Однако после оценки условия эта временная переменная будет уничтожена, так что дальнейшее применение *cs* под вопросом.

Причина, по которой в рассмотренном примере (и во многих других подобных примерах) возникла проблема с временной переменной, заключается в том, что высокоуровневый тип данных используется в несвойственном ему низкоуровневом стиле. Более адекватный стиль программирования не только позволяет избежать рассмотренных проблем с временными переменными, но и порождает существенно более понятный код. Например:

```
void f(string& s1, string& s2, string& s3)
{
  cout<< s1+s2;
  string s = s2+s3;

  if(s.length()<8 && s[0]=='a')
  {
    // s используется здесь
  }
}
```

Временные объекты можно использовать в качестве инициализаторов для *константных* ссылок или именованных объектов. Например:

```
void g(const string&, const string&);

void h(string& s1, string& s2)
{
  const string& s = s1+s2;
  string ss = s1+s2;
  g(s,ss);                    // здесь можно использовать s и ss
}
```

Здесь все корректно. Временный объект уничтожается только тогда, когда «его» ссылка или именованный объект выходят из области видимости. Помните, что возврат ссылки на локальную переменную является ошибкой (§7.3), и что временный объект не может связываться с *неконстантной* ссылкой (§5.5).

Временный объект можно создавать прямым вызовом конструктора. Например:

```
void f(Shape& s, int x, int y)
{
  s.move(Point(x,y));        // создается Point для передачи Shape::move()
  // ...
}
```

Такие временные объекты уничтожаются точно так же, как и временные объекты, создаваемые неявно.

10.4.11. Размещение объектов в заданных блоках памяти

С помощью операции ***new*** объекты по умолчанию создаются в свободной памяти. А что, если мы захотим разместить объект где-либо в другом месте? Рассмотрим простой класс:

```
class X
{
public:
```

```
  X(int);
  // ...
};
```

Мы можем разместить динамически создаваемый объект где-угодно, если определим функцию выделения памяти ***operator new***() с дополнительным (вторым) формальным параметром, фактическое значение которого будет задаваться при помощи дополнительного параметра в операции ***new***:

```
void* operator new(size_t, void* p) {return p;}  // см. (§19.4.5)
void* buf = reinterpret_cast<void*>(0xF00F);     // некоторый значимый адрес

X* p2=new (buf) X;    // вызывается функция operator new (sizeof(X), buf)
                      // и конструируется объект X в буфере buf
```

Из-за такого применения вариант операции new в виде ***new*** (***buf***) ***X***, предназначенный для передачи дополнительного аргумента функции ***operator new()***, называется операцией «*размещающее new*» (*placement new*). Обратите внимание на то, что первым аргументом функции ***operator new***() всегда является размер выделяемого блока памяти, и что размер размещаемого объекта передается неявно (§15.6). Выбор конкретного варианта функции ***operator new***() для соответствия применяемому варианту операции ***new*** выполняется, как всегда, по правилу соответствия аргументов (§7.4); при этом первый аргумент каждого варианта функции ***operator new***() должен иметь тип ***size_t***.

Приведенный пример функции ***operator new***() является простейшим примером распределителя памяти для операции «размещающее ***new***». Он определен в стандартном заголовочном файле <***new***>.

В рассмотренном примере мы использовали операцию приведения ***reinterpret_cast***, наиболее «безобразную на вид» и опасную из всех операций приведения (§6.2.7). В большинстве случаев эта операция оставляет последовательность бит операнда нетронутой, изменяя лишь трактовку его типа. Ее применяют, например, в потенциально опасных, но иногда необходимых низкоуровневых системнозависимых преобразованиях целочисленных значений в указатели и наоборот.

Операцию «размещающее ***new***» можно использовать для помещения объектов в конкретные участки памяти:

```
class Arena
{
public:
  virtual void* alloc(size_t) =0;
  virtual void free(void*) =0;
  // ...
};

void* operator new(size_t sz, Arena* a)
{
  return a->alloc(sz);
}
```

Теперь объекты произвольных типов можно при необходимости размещать в тех или иных специфических участках памяти («*аренах*»). Например:

```
extern Arena* Persistent;
extern Arena* Shared;

void g (int i)
{
  X* p = new (Persistent) X(i) ;    // X в арене Persistent
  X* q = new (Shared) X(i) ;        // X в арене Shared
  // ...
}
```

Помещение объекта в область памяти, не находящуюся под управлением стандартного механизма выделения/освобождения свободной памяти, подразумевает специальные действия при уничтожении объекта. Базовым механизмом является в этом случае *явный вызов деструктора*:

```
void destroy (X* p, Arena* a)
{
  p->~X() ;                         // вызов деструктора
  a->free(p) ;                      // освобождение памяти
}
```

Отметим, что явный вызов деструкторов и применение специальных вариантов *глобальных* аллокаторов памяти в обычных случаях нецелесообразны. Все же, в определенных случаях они нужны. Например, трудно было бы реализовать обобщенный контейнер в духе стандартного библиотечного класса **vector** (§3.7.1, §16.3.8) без применения явного вызова деструктора. Новичкам лучше трижды подумать перед тем, как явно вызвать деструктор, а еще лучше проконсультироваться с опытным специалистом.

В §14.4.4 рассмотрено, как исключения взаимодействуют с операциями «размещающее **new**».

Не существует отдельного синтаксиса для размещения массивов. В этом нет нужды, так как операция «размещающее **new**» и так работает с произвольными типами данных. Тем не менее, для массивов можно определить специальный вариант функции **operator delete**() (§19.4.5).

10.4.12. Объединения

Именованное объединение определяется как *структура*, в которой всем полям отводится одна и та же область памяти (§C.8.2). Объединение может иметь функции-члены, но не статические члены[1].

В общем случае, компилятор не может знать, какой член объединения используется; то есть тип объекта, хранящегося в объединении, неизвестен. Следовательно, объединение не может содержать члены, классы которых имеют явно определенные конструктор или деструктор. Иначе не будет возможности предотвратить хранящийся в объединении объект от непреднамеренной порчи, и невозможно будет вызвать правильный деструктор при выходе объединения из области видимости.

---

[1] Подробнее этот вопрос рассматривается в книге Н.Н. Мартынов, «Программирование для Windows на C/C++», том 2, М., Бином, 2006, стр. 67—69. — *Прим. ред.*

Лучше всего ограничить использование объединений низкоуровневым кодом, или в виде части класса, содержащего информацию о том, что на самом деле хранится в объединении (см. §10.6[20]).

10.5. Советы

1. Представляйте концепции в виде классов; §10.1.
2. Используйте открытые данные (*структуры*) только тогда, когда это на самом деле лишь данные, и для них не существует разумных инвариантов; §10.2.8.
3. Конкретные типы являются простейшими видами классов. Где только возможно, предпочитайте конкретные типы более сложным видам классов или открытым структурам данных; §10.3.
4. Определяйте функцию в качестве функции-члена класса только в тех случаях, когда ей нужен непосредственный доступ к его внутренней структуре; §10.3.2.
5. Используйте пространства имен для явного выражения связи класса и его функций поддержки; §10.3.2.
6. Объявляйте функции-члены *константными*, если они не модифицируют состояния (значения) объектов; §10.2.6.
7. Объявляйте функции-члены *статическими* в тех случаях, когда им нужен доступ к представлению класса, но не нужен вызов для конкретных объектов класса; §10.2.4.
8. Используйте конструктор для установления инварианта класса; §10.3.1.
9. Если конструктор класса выделяет ресурсы, то для освобождения ресурсов классу требуется деструктор; §10.4.1.
10. Если в классе содержится член-указатель, то для класса следует определить копирующие операции (копирующий конструктор и операцию присваивания); §10.4.4.1.
11. Если класс содержит член-ссылку, то ему, возможно, следует определить копирующие операции (копирующий конструктор и операцию присваивания); §10.4.6.3.
12. Если классу требуются копирующие операции или деструктор, то ему вероятно потребуются конструктор, деструктор, операция присваивания и копирующий конструктор; §10.4.4.1.
13. В операции присваивания проверяйте возможность самоприсваиваний; §10.4.4.1.
14. При написании копирующего конструктора следите за тем, чтобы были скопированы все элементы (остерегайтесь умолчательной инициализации); §10.4.4.1.
15. Добавляя новый член к существующему классу, проверяйте необходимость модификации конструкторов с целью корректной инициализации новых членов; §10.4.6.3.

16. Пользуйтесь перечислениями для объявлений целых констант в классе; §10.4.6.2.
17. Избегайте зависимости от порядка конструирования глобальных объектов и объектов в пространствах имен; §10.4.9.
18. Для минимизации зависимости от порядка конструирования объектов применяйте флаги первого использования; §10.4.9.
19. Помните, что временные объекты уничтожаются в конце вычисления полных выражений, в которых они созданы; §10.4.10.

10.6. Упражнения

1. (*1) Найдите ошибку в ***Date*::*add_year*()** в §10.2.2. Затем найдите еще две ошибки в версии из §10.2.7.
2. (*2.5) Скомпилируйте и проверьте работу ***Date***. Переработайте этот класс для представления даты в виде «количество дней после 1/1/1970».
3. (*2) Найдите какой-либо коммерческий вариант типа ***Date***. Покритикуйте предоставляемые им возможности. Обсудите это с другими пользователями.
4. (*1) Как вы обратитесь к ***set_default*** из класса ***Date***, находящегося в пространстве имен ***Chrono*** (§10.3.2). Предложите по крайней мере три разных способа.
5. (*2) Определите класс ***Histogram***, который хранит числа из интервалов, указанных аргументами конструктора. Напишите функции для вывода гистограмм. Обрабатывайте ошибки выхода из диапазона значений.
6. (*2) Определите несколько классов для представления случайных чисел с разными законами распределения (например, равномерным или экспоненциальным). В каждом классе должен быть конструктор, задающий параметры распределения, и функция ***draw***(), возвращающая следующее случайное число.
7. (*2.5) Дополните класс ***Table*** возможностью хранения пар (имя, значение). Затем модифицируйте программу-калькулятор из §6.1 таким образом, чтобы она использовала класс ***Table*** вместо ***map***. Сравните две версии.
8. (*2) Перепишите ***Tnode*** из §7.10[7] в виде класса с конструкторами, деструктором и т.д. Определите дерево с узлами ***Tnode*** в виде класса с конструкторами, деструктором и т.д.
9. (*3) Определите, реализуйте и протестируйте класс ***Intset*** (множество целых). Предоставьте операции объединения множеств, пересечения и симметричной разности.
10. (*1.5) Модифицируйте класс ***Intset*** во множество узлов ***Node***, где ***Node*** — определяемая вами структура.
11. (*3) Определите класс для анализирования, хранения, вычисления и печати простых арифметических выражений, состоящих из целых констант и операций +, -, * и /. Открытый интерфейс должен выглядеть так:

```
class Expr
{
// ...
public:
  Expr (const char*) ;
  int eval () ;
  void print () ;
};
```

Строковый аргумент конструктора *Expr::Expr()* является выражением. Функция *Expr::eval()* возвращает значение выражения, а *Expr::print()* выводит представление выражения в *cout*. Программа может выглядеть примерно так:

```
Expr x ("123/4+123*4-3") ;
cout<<"x= "<<x.eval()<<"\n";
x.print () ;
```

Определите класс *Expr* дважды: один раз для его представления выберите список связных узлов, а другой раз — строку. Поэкспериментируйте с разными способами печати выражения: с полной расстановкой скобок, в постфиксной нотации и т.д.

12. (*2) Определите класс *Char_queue* так, чтобы открытый интерфейс не зависел от представления. Реализуйте *Char_queue* в виде (a) связного списка и (b) вектора. Не беспокойтесь о многопоточности.

13. (*3) Разработайте класс таблицы символов и класс элементов этой таблицы для какого-либо языка. Посмотрите на какой-нибудь компилятор для этого языка, чтобы узнать, как в действительности выглядит таблица символов.

14. (*2) Модифицируйте класс выражений из §10.6[11] таким образом, чтобы он мог обрабатывать переменные и операцию присваивания. Воспользуйтесь классом таблицы символов из §10.6[13].

15. (*1) Дана программа:

```
#include <iostream>

int main ()
{
  std::cout<<"Hello, world!\n";
}
```

Модифицируйте ее так, чтобы она выводила

```
Initialize
Hello, world!
Clean up
```

Не вносите при этом никаких изменений в функцию *main()*.

16. (*2) Определите класс *Calculator* так, чтобы большая часть его реализации состояла из функций §6.1. Создавайте калькуляторы и активизируйте их для ввода из *cin*, из командной строки, для строк программы. Реализуйте вывод в разные приемники.

17. (*2) Определите два класса, каждый со ***статическим*** членом, так, чтобы конструирование ***статического*** члена использовало ссылку на другой ***статический*** член. Где такое может встретиться в реальном коде? Как нужно модифицировать эти классы, чтобы устранить в конструкторах зависимость от порядка?

18. (*2.5) Сравните класс ***Date*** (§10.3) с вашими решениями упражнений §5.9[13] и §7.10[19]. Обсудите найденные ошибки и возможные различия в сопровождении каждого из решений.

19. (*3) Напишите функцию, которая получает в качестве аргументов ***istream*** и ***vector***<***string***>, а порождает ***map***<***string***, ***vector***<***int***>>, содержащий каждую строку и частоту ее вхождения. Прогоните программу на текстовом файле с количеством строк, не менее 1000, разыскивая при этом не менее 10 слов.

20. (*2) Возьмите класс ***Entry*** из §C.8.2 и модифицируйте его таким образом, чтобы каждый член объединения всегда использовался в соответствии с его типом.

11

Перегрузка операций

Когда я использую слово,
оно означает то, что я хочу им выразить,
не больше и не меньше.
— Шалтай-Болтай

Нотация операций —функции-операции — бинарные и унарные операции — предопределенный смысл операций —операции и пользовательские типы — операции и пространства имен — комплексный тип — операции в виде функций-членов и глобальных функций — смешанная арифметика — инициализация — копирование — преобразования — литералы — функции поддержки — операции приведения типов — разрешение неоднозначности — друзья — члены и друзья —объекты больших размеров — присваивание и инициализация — индексирование — операция вызова функции — разыменование — инкремент и декремент — класс строк — советы — упражнения.

11.1. Введение

В любой технической области — и в большинстве не технических — вырабатываются свои стандартные обозначения, помогающие наглядно представлять и обсуждать часто используемые концепции. Например, из-за длительного знакомства с формой записи

*x+y*z*

она для нас намного яснее и понятнее, чем фраза

умножить y на z и прибавить результат к x

Невозможно переоценить важность кратких нотаций для общепринятых операций.

Как и большинство других языков программирования, C++ поддерживает набор операций для встроенных типов. Однако большинство концепций, для которых традиционно используются операции, не относятся ко встроенным типам языка C++, и их нужно представлять пользовательскими типами. Например, если вам

нужны комплексная арифметика, матричная алгебра или символьные строки, в языке C++ вы используете классы для представления этих концепций. А если для этих типов данных определить смысл действия стандартных операций, то работа пользователя с объектами этих классов становится более простой и наглядной по сравнению с использованием лишь традиционной функциональной нотации. Например,

```
class complex               // очень упрощенный класс complex
{
  double re, im;

public:
  complex(double r, double i) : re(r), im(i) { }
  complex operator+ (complex) ;
  complex operator* (complex) ;
};
```

реализует концепцию комплексных чисел. *Комплексное* число представлено здесь в виде набора двух вещественных чисел и дополнено арифметическими операциями + и * над этим новым типом данных. Программист в классе complex определяет функции-члены ***operator+()*** и ***operator*()***, которые и задают смысл и порядок работы операций + (сложение) и * (умножение) над объектами типа ***complex***. Если ***b*** и ***c*** имеют тип complex, то ***b+c*** означает функциональный вызов ***b.operator+(c)***. Теперь для типа ***complex*** мы можем применять нотацию, принятую в математической литературе для комплексных чисел:

```
void f()
{
  complex a = complex(1,3.1) ;
  complex b = complex(1.2,2) ;
  complex c = b;

  a = b+c;
  b = b+c*a;
  c = a*b+complex(1,2) ;
}
```

При этом сохраняются обычные правила приоритета операций, так что второе выражение в представленном примере означает ***b=b+(c*a)***, а не ***b=(b+c)*a***.

Наиболее очевидные случаи перегрузки операций (то есть переопределения операций для нового типа) относятся к конкретным типам данных (§10.3). Но, разумеется, полезность определяемых пользователем операций выходит за эти рамки. Например, при проектировании общих и специализированных интерфейсов программных систем часто приходится переопределять поведение даже таких операций, как ->, [] и ().

11.2. Функции-операции

Можно перегружать следующие операции:

```
+      -      *      /      %       ^       &
|      ~      !      =      <       >       +=
-=     *=     /=     %=     ^=      &=      |=
<<     >>     >>=    <<=    ==      !=      <=
>=     &&     ||     ++     --      ->*     ,
->     []     ()     new    new[]   delete  delete[]
```

Нельзя перегружать операции:

- `::` (разрешение области видимости; §4.9.4, §10.2.4),
- `.` (выбор члена класса; §5.7) и
- `.*` (выбор члена класса через указатель на классовые члены; §15.5).

Правым операндом у этих операций является имя, а не значение, и обеспечивают они базовое средство доступа к членам. Возможность перегрузки этих операций привела бы к очень тонким ошибкам [Stroustrup, 1994]. Тернарная условная операция `?:` (§6.3.2) также запрещена к перегрузке. Также нельзя перегружать именованные операции ***sizeof*** (§4.6) и ***typeid*** (§15.4.4).

Невозможно использовать не существующие в языке C++ знаки операций, но можно воспользоваться функциями в случаях, когда существующие знаки операций не подходят. Например, используйте ***pow***`()` вместо `**`[1]. Такие ограничения могут показаться драконовскими, но более гибкая система обозначений легко приводит к неоднозначностям. Например, определение новой операции `**` для обозначения возведения в степень кажется на первый взгляд очевидной и несложной в реализации, но это только на первый взгляд. Действительно, должна ли операция `**` быть левоассоциативной (как в языке Fortran), или правоассоциативной (как в язке Algol)? Как должно интерпретироваться выражение `a**p` — как `a*(*p)`, или как `(a)**(p)`?

Имя *функции-операции* (*operator function*) начинается с ключевого слова ***operator***, за которым следует знак перегружаемой операции, например ***operator***`<<`. Функция-операция объявляется и может быть вызвана так же, как и любая другая функция. Традиционная форма использования операции является сокращением явной функциональной формы вызова. Например:

```
void f(complex a, complex b)
{
  complex c = a + b;                 // сокращенная форма
  complex d = a.operator+(b);        // явный вызов
}
```

Оба инициализирующих выражения в данном примере эквивалентны друг другу.

[1] Речь идет об операции возведения в степень. — *Прим. ред.*

11.2.1. Бинарные и унарные операции

Бинарную (двухоперандную) операцию можно перегрузить либо с помощью нестатической функции-члена с одним аргументом, либо с помощью глобальной функции с двумя аргументами. При этом для любой бинарной операции @ выражение ***aa@bb*** интерпретируется либо как ***aa.operator@(bb)***, либо как ***operator@(aa,bb)***. Если определены оба варианта, то для выбора одного из них используются критерии разрешения перегрузки (§7.4). Например:

```
class X
{
public:
  void operator+ (int) ;
  X(int) ;
};

void operator+ (X,X) ;
void operator+ (X,double) ;

void f(X a)
{
  a+1;                    // a.operator+(1)
  1+a;                    // ::operator+(X(1),a)
  a+1.0;                  // ::operator+(a,1.0)
}
```

Унарную (однооперандную; префиксную или постфиксную) операцию можно определить либо с помощью нестатической функции-члена без аргументов, либо с помощью глобальной функции с одним аргументом. Для любой префиксной унарной операции @ выражение ***@aa*** интерпретируется либо как ***aa.operator@()***, либо как ***operator@(aa)***. Если определены оба варианта, то для выбора одного из них используются критерии разрешения перегрузки (§7.4). Для любой постфиксной унарной операции @ выражение ***aa@*** интерпретируется либо как ***aa.operator@(int)***, либо как ***operator@(aa,int)***. Более подробно это объясняется в §11.11. Если определены оба варианта, то для выбора одного из них используются критерии разрешения перегрузки (§7.4). Любая операция при перегрузке должна объявляться в соответствии с ее стандартным синтаксисом (§A.5). Например, пользователь не может определить унарную операцию % или тернарную операцию +. Рассмотрим пример:

```
class X
{
public:
  // members (имеющие неявный указатель this):
  X* operator& () ;       // префиксная унарная & (взятие адреса)
  X operator& (X) ;       // бинарная & ("И")
  X operator++ (int) ;    // постфиксный инкремент (см. §11.11)
  X operator& (X,X) ;     // error: три операнда (тернарная операция)
  X operator/ () ;        // error: унарная операция /
};

// nonmember functions:
X operator- (X) ;         // префиксный унарный минус
```

```
X operator- (X,X) ;       // бинарный минус
X operator-- (X&,int) ;   // постфиксный декремент
X operator- () ;          // error: нет операнда
X operator- (X,X,X) ;     // error: тернарная операция
X operator% (X) ;         // error: унарная операция %
```

Перегрузка операции [] рассматривается в §11.8, операции () — в §11.9, операции -> — в §11.10, операций -- и ++ — в §11.11; перегрузка операций динамического выделения памяти и ее освобождения рассматривается в §6.2.6.2, §10.4.11 и §15.6.

11.2.2. Предопределенный смысл операций

Относительно перегружаемых пользователем операций существует лишь ряд дополнительных правил. В частности, **operator=** (), **operator**[] (), **operator**() () и **operator->** () *обязаны быть нестатическими функциями-членами* (и только ими); это гарантирует, что их первый операнд всегда lvalue (§4.9.6).

Для некоторых встроенных операций постулируется их эквивалентность определенной комбинации других операций. Например, если a имеет тип *int*, то *++a* означает то же, что и *a += 1*, что также эквивалентно *a=a+1*. Такие соотношения между операциями не поддерживаются для перегружаемых пользователем операций автоматическим образом, а только если пользователь позаботился об этом сам. Например, компилятор не генерирует автоматически определение для *Z*: : **operator+=** () из одного того факта, что пользователь определил *Z*: : **operator+** () и *Z*: : **operator=** ().

Исторически так сложилось, что операции = (операция присваивания), & (операция взятия адреса) и , (операция следования; §6.2.2) имеют предопределенный смысл, когда применяются к классовым объектам. Их можно сделать недоступными для пользователей класса, если объявить операции закрытыми:

```
class X
{
private:
  void operator= (const X&) ;
  void operator& () ;
  void operator, (const X&) ;
  // ...
};

void f(X a, X b)
{
  a = b;          // error: operator= private
  &a;             // error: operator& private
  a,b;            // error: operator, private
}
```

А можно придать им новый смысл, переопределив их в классе соответствующим образом (естественно, в открытом режиме).

11.2.3. Операции и пользовательские типы

Любая *функция-операция* должна быть либо *членом класса, либо иметь по крайней мере один аргумент классового типа* (за исключением функций **operator new**() и **operator delete**(), участвующих в перегрузке операций *new*/*delete*). Это правило га-

рантирует, что пользователь не может изменить смысл выражения, если оно не содержит объектов пользовательских типов. В частности, нельзя изменить смысл операций, выполняющихся над указателями. Можно сказать, что язык C++ является расширяемым, но он не подвержен мутациям (за исключением операций =, & и , для классовых объектов).

Функция-операция, принимающая первый аргумент встроенного типа (§4.1.1), не может быть функцией-членом класса. Например, рассмотрим сложение комплексной переменной aa и целого числа *2*: выражение ***aa+2*** может при наличии соответствующего определения функции-члена рассматриваться как ***aa.operator+(2)***, но выражение ***2+aa*** не может никак интерпретироваться, ибо нет класса ***int***, для которого была бы определена операция + с возможность трактовки выражения как ***2.operator+(aa)***. Но даже, если бы такой класс и был, то все равно выражения ***aa+2*** и ***2+aa*** требовали бы для интерпретации двух разных функций-членов. Из-за того, что компилятор не понимает смысла определяемых пользователем операций, он не может понять, что операция + коммутативна, и интерпретировать ***2+aa*** как ***aa+2***. Рассмотренная проблема легко решается с помощью определения глобальных функций-операций (§11.3.2, §11.5).

Так как *перечисления являются пользовательскими типами*, то для них *можно определять операции*. Например:

```
enum Day {sun, mon, tue, wed, thu, fri, sat};

Day& operator++(Day& d)
{
  return d = (sat==d) ? sun:Day(d+1);
}
```

Любое выражение проверяется на отсутствие неоднозначностей. Когда имеются предоставляемые пользователем операции, неоднозначности в выражениях устраняются по правилам из §7.4.

11.2.4. Операции и пространства имен

Операции определяются либо в классах, либо в пространствах имен (в частности, в глобальном пространстве). Рассмотрим упрощенную версию ввода/вывода строк из стандартной библиотеки:

```
namespace std          // упрощенное std
{
  class ostream
  {
    // ...
    ostream& operator<<(const char*);
  };

  extern ostream cout;

  class string
  {
    // ...
  };
```

```
  ostream& operator<< (ostream&, const string&) ;
}

int main ()
{
  char* p = "Hello";
  std::string s = "world";
  std::cout<< p << ", " << s << "!\n";
}
```

Естественно, представленный код выводит строку ***Hello, world***! Но почему? Заметьте, что я не стал делать доступными все средства из пространства имен ***std*** при помощи директивы

```
using namespace std;
```

Вместо этого я воспользовался префиксом ***std::*** для ***string*** и ***cout***. Таким образом я не стал засорять глобальное пространство имен и порождать ненужные зависимости.

Для вывода С-строк (то есть ***char****) в классе ***std::ostream*** имеется соответствующая функция-операция, так что по определению

```
std::cout<< p
```

означает

```
std::cout.operator<< (p)
```

Но для вывода строк типа ***std::string*** в классе ***std::ostream*** соответствующей функции-члена нет, так что

```
std::cout<< s
```

означает

```
operator<< (std::cout, s)
```

Операции, определенные в пространствах имен, различаются по типам операндов точно так же, как функции различаются по типам их аргументов (§8.2.6). В частности, так как cout определен в пространстве имен ***std***, то это пространство имен включается в процесс поиска подходящего определения для операции <<. В результате, компилятор находит и использует

```
std::operator<< (std::ostream&, const std::string&)
```

Рассмотрим бинарную операцию @. Если x имеет тип ***X***, а y имеет тип ***Y***, то для выражения ***x@y*** выполняется следующий поиск определений:

- Если ***X*** класс, искать определение функции-члена ***operator@*** в классе ***X*** или в его базовых классах.
- Искать объявление ***operator@*** в контексте, окружающем ***x@y***.
- Если ***X*** определен в пространстве имен ***N***, искать объявление ***operator@*** в ***N***.
- Если ***Y*** определен в пространстве имен ***M***, искать объявление ***operator@*** в ***M***.

Если находятся несколько подходящих операций, то для выбора наилучшего соответствия (если оно существует) применяются критерии разрешения перегрузки

(§7.4). Рассмотренный механизм поиска выполняется только в том случае, когда один из операндов операции имеет пользовательский тип; при этом будут учтены операции приведения типа, определенные пользователем (§11.3.2, §11.4). Обратите внимание на то, что имя, введенное оператором ***typedef***, является лишь синонимом, а не пользовательским типом (§4.9.7).

Поиск определений унарных операций и разрешение неоднозначностей выполняются аналогично.

Заметьте, что при поиске определений для операций никакого преимущества у функций-членов перед глобальными функциями нет; в этом состоит отличие от поиска именованных функций (§8.2.6). *Меньшее сокрытие операций* приводит к тому, что встроенные операции всегда доступны и что пользователи могут придавать операциям новый смысл без модификации существующих определений классов. Например, в стандартной библиотеке операция вывода << определена по отношению ко встроенным типам. Пользователь же всегда может доопределить эту операцию для вывода своих собственных классовых типов без какой бы то ни было модификации класса ***ostream*** (§21.2.1).

11.3. Тип комплексных чисел

Реализация концепции комплексных чисел, представленная во введении, слишком ограничена, чтобы подойти для реальной работы. Например, заглянув в книгу по математике, мы ожидаем, что следующее будет работать:

```
void f()
{
  complex a = complex(1,2);
  complex b = 3;
  complex c = a+2.3;
  complex d = 2+b;
  complex e = -b-c;
  b = c*2*c;
}
```

Кроме того, мы ожидаем, что реализованы такие операции, как операция == для сравнения комплексных чисел и операция << для вывода, а также набор подходящих математических функций, таких как ***sin***() и ***sqrt***().

Класс ***complex*** является конкретным типом, так что будем следовать рекомендациям из §10.3. Кроме того, поскольку стандартная комплексная арифметика опирается на применение операций, то нам придется в рамках типа complex использовать все, что уже известно про перегрузку операций.

11.3.1. Перегрузка операций функциями-членами и глобальными функциями

Я предпочитаю минимизировать количество функций, непосредственно манипулирующих внутренним представлением объекта класса. Для этого в теле класса следует определять лишь те операции, которые по своей внутренней природе должны изменять состояние первого операнда, например +=. Любую иную операцию,

целью которой является порождение нового значения, операцию + к примеру, следует реализовывать вне класса (при этом она может опираться на фундаментальные операции, имеющие доступ к представлению класса):

```
class complex
{
  double re, im;
public:
  complex& operator+= (complex a) ; // имеет доступ к внутреннему представлению
  // ...
};

complex operator+ (complex a, complex b)
{
  complex r = a;
  return r+= b;                    // получает доступ к представлению через +=
}
```

Отталкиваясь от данных определений, мы можем писать:

```
void f(complex x, complex y, complex z)
{
  complex r1 = x+y+z;              // r1 = operator+(x,operator+(y,z))
  complex r2 = x;                  // r2 = x
  r2 += y;                         // r2.operator+=(y)
  r2 += z;                         // r2.operator+=(z)
}
```

Составные операции присваивания, такие как += и *=, программировать легче, чем их более «простые составные части» — операции + и *. Поначалу это многих удивляет, но это вытекает из того факта, что три объекта вовлекаются в работу операции + (два операнда и временный объект-результат), в то время как у операции += — всего два объекта. В последнем случае достигается более высокая эффективность за счет устранения необходимости во временных объектах. Например, следующий код

```
inline complex& complex::operator+= (complex a)
{
  re += a.re;
  im += a.im;
  return *this;
}
```

не требует временных объектов для хранения результатов сложения и упрощает компилятору задачу по реализации встраиваемого кода.

Оптимизирующий компилятор сумеет сгенерировать код, близкий к оптимальному, и для операции +. К сожалению, не каждый компилятор может похвастаться блестящими оптимизирующими способностями, и не каждый пользовательский тип столь же прост как тип complex; поэтому в §11.5 обсуждаются синтаксические средства, позволяющие операциям иметь прямой доступ к представлению класса.

11.3.2. Смешанная арифметика.

Чтобы справиться с выражением

```
complex d = 2 + b;
```

нам нужно определить операцию +, работающую с операндами разных типов. В терминологии языка Fortran, нам требуется *смешанная арифметика* (*mixed-mode arithmetic*). Этого можно добиться путем добавления всех нужных версий операции сложения:

```
class complex
{
  double re, im;
public:
  complex& operator+= (complex a)
  {
    re += a.re;
    im += a.im;
    return *this;
  }
  complex& operator+= (double a)
  {
    re += a;
    return *this;
  }
  // ...
};

complex operator+ (complex a, complex b)
{
  complex r = a;
  return r += b;                  // вызывает complex::operator+=(complex)
}

complex operator+ (complex a, double b)
{
  complex r = a;
  return r += b;                  // вызывает complex::operator+=(double)
}

complex operator+ (double a, complex b)
{
  complex r = b;
  return r += a;                  // вызывает complex::operator+=(double)
}
```

Операция сложения с числом типа ***double*** проще, чем сложение двух *комплексных* чисел, что и отражается в коде данного примера. Операции с ***double*** не затрагивают мнимых частей *комплексных* чисел, и поэтому более эффективны.

Отталкиваясь от имеющихся определений, мы можем писать, например, такой клиентский код:

```
void f(complex x, complex y)
{
  complex r1 = x+y;          // вызывает operator+(complex,complex)
  complex r2 = x+2;          // вызывает operator+(complex,double)
  complex r3 = 2+x;          // вызывает operator+(double,complex)
}
```

11.3.3. Инициализация

Чтобы иметь возможность копирования и инициализации *комплексных* чисел вещественными скалярами, нам требуется преобразование из такого скаляра в комплексное число. Например:

```
complex b = 3;               // должно означать b.re=3, b.im=0
```

Конструктор, принимающий *единственный аргумент*, осуществляет *преобразование* (приведение) *типа аргумента к классовому типу*. Например:

```
class complex
{
  double re, im;

public:
  complex(double r) : re(r), im(0) { }
  // ...
};
```

Конструктор предписывает, как сформировать значение классового типа. Конструктор используется в случаях, когда ожидается значение такого типа и когда оно может быть создано конструктором из значений, предоставляемых инициализатором, или из присваиваемых значений. Поэтому *конструктор с единственным аргументом вызывать явно не нужно.* Например,

```
complex b = 3;
```

означает

```
complex b = complex(3);
```

Неявное применение определяемого пользователем приведения типов возможно лишь в случае, если оно уникально (§7.4). В §11.7.1 рассматривается способ определения конструктора, для которого неявный вызов запрещен.

Естественно, нам обязательно нужен конструктор с двумя аргументами типа ***double***, а также умолчательный конструктор для построения комплексного числа **(*0, 0*)**:

```
class complex
{
  double re, im;

public:
  complex() : re(0), im(0) { }
  comlex(double r) : re(r), im(0) { }
  comlex(double r, double i) : re(r), im(i) { }
  // ...
};
```

Применяя аргументы по умолчанию, мы можем добиться некоторого сокращения кода:

```
class complex
{
  double re, im;

public:
  complex(double r =0, double i =0) : re(r), im(i) { }
  //...
};
```

При наличии в классе явно определенных конструкторов использование списков инициализации (§5.7, §4.9.5) запрещается. Например:

```
complex z1 = {3};        // error: у типа complex есть конструктор
complex z2 = {3,4};      // error: у типа complex есть конструктор
```

11.3.4. Копирование

В дополнение к определенным в классе *конструкторам*, класс имеет умолчательный вариант (неявно генерируемый компилятором) копирующего конструктора (§10.2.5), который просто копирует все поля данных. То же самое можно явным образом определить и самим:

```
class complex
{
  double re, im;

public:
  complex(const complex& c) : re(c.re), im(c.im) { }
  //...
};
```

В тех случаях, когда умолчательный вариант копирующего конструктора подходит, я предпочитаю ничего самому не писать. Это лаконичнее и люди заранее понимают его работу. Кроме того, компилятору удобно оптимизировать свое собственное творение. И, наконец, ручное кодирование простейшего копирования полей данных утомительно и чревато внесением нелепых ошибок, особенно для классов со многими полями данных (§10.4.6.3).

Для *копирующего конструктора* я использую *аргумент, передаваемый по ссылке*, и я просто *обязан* это делать, поскольку копирующий конструктор определяет, как проходит процесс копирования, в том числе аргументов этого типа. Таким образом, *передача аргумента копирующему конструктору по значению*

```
complex::complex(complex c) : re(c.re), im(c.im) { }    // error
```

приводит к бесконечной рекурсии.

Для остальных же функций я использую передачу аргументов типа **complex** по значению, а не по ссылке. Здесь у разработчика есть выбор. С точки зрения пользователя разница между вариантами аргументов типа **complex** или const **complex&** невелика. Этот вопрос обсуждается далее в §11.6.

В принципе, копирующие конструкторы используются в простых случаях инициализации вроде

```
complex x = 2;                    // создается complex(2) и им инициализируется x
complex y = complex (2, 0) ;      // создается complex(2,0) и им инициализируется y
```

В то же время, вызовы копирующих конструкторов могут быть легко заменены на эквивалентные варианты

```
complex x (2) ;                   // x инициализируется значением 2
complex y (2, 0) ;                // x инициализируется парой чисел (2,0)
```

Для арифметических типов я предпочитаю вариант с использованием =. Можно ограничить набор допустимых для стиля инициализации со знаком = значений по сравнению с вариантом инициализации, использующим (), сделав копирующий конструктор закрытым (§11.2.2) или объявив его с модификатором ***explicit*** (§11.7.1).

Аналогично инициализации, присваивание по умолчанию одного объекта класса другому эквивалентно почленному присваиванию (§10.2.5). Мы могли бы и явным образом задать такое действие с помощью ***complex*::*operator*=** (). Однако для простых типов, таких как ***complex***, в этом нет никакой нужды. Достаточно умолчательного варианта.

Копирующий конструктор — и умолчательный, и пользовательский — *используется не только при инициализации переменных*, но и *при передаче аргументов*, *возврате значений*, а также *обработке исключений* (§11.7). Семантика этих операций эквивалентна семантике инициализации (§7.1, §7.3, §14.2.1).

11.3.5. Конструкторы и преобразования типов

Мы определили по три версии для каждой из четырех арифметических операций:

```
complex operator+ (complex, complex) ;
complex operator+ (complex, double) ;
complex operator+ (double, complex) ;
// ...
```

Такое дублирование довольно утомительно, а что утомительно, то чревато ошибками. А что, если было бы по три альтернативы для каждого аргумента каждой функции. Тогда потребовались бы три версии для каждой одноаргументной функции, девять версий каждой двухаргументной функции, двадцать семь версий для каждой трехаргументной функции и т.д. Часто, все эти варианты похожи друг на друга — они сначала приводят аргументы к общему типу, а потом выполняют один и тот же стандартный алгоритм.

Альтернатива такому множественному определению функций для каждой из возможных комбинаций типов аргументов может базироваться на автоматических преобразованиях аргументов. Например, наш класс ***complex*** имеет конструктор, который преобразовывает ***double*** в ***complex***. В результате мы можем определить единственную версию операции сравнения:

```
bool operator== (complex, complex) ;

void f(complex x, complex y)
{
  x==y;                          // означает operator==(x,y)
  x==3;                          // означает operator==(x,complex(3))
  3==y;                          // означает operator==(complex(3),y)
}
```

Конечно, могут существовать причины для предпочтения варианта с несколькими отдельными функциями. Например, в каких-то случаях преобразования типов аргументов могут оказаться обременительными с точки зрения производительности, а в других случаях — может применяться более простой и эффективный алгоритм для специфических аргументов. Во всех остальных случаях целесообразно предоставить один наиболее общий вариант функции, базирующийся на преобразованиях типов аргументов, плюс, возможно, несколько критических вариантов, и это предотвращает *комбинаторный взрыв вариантов, типичный для смешанной арифметики.*

При наличии нескольких вариантов функций (операций) компилятор должен выбрать наиболее подходящий вариант, основываясь на типах аргументов (операндов) и доступных преобразованиях (стандартных или определяемых пользователем). Если невозможно выбрать единственный наилучший вариант, то выражение трактуется как ошибочное (неоднозначное) (см. §7.4).

Объект в выражении, созданный явно или неявно конструктором, является автоматическим и будет уничтожен при первой же возможности (см. §10.4.10).

Пользовательские преобразования не применяются неявным образом к левым операндам операций . или ->. Это справедливо и для случаев, когда операция . подразумевается (хотя явно и не пишется):

```
void g (complex z)
{
  3+z;                           // ok: complex(3)+z
  3.operator+= (z) ;             // error: 3 не является объектом класса
  3+=z;                          // error: 3 не является объектом класса
}
```

Таким образом, если операция в качестве левого операнда требует lvalue, то ее нужно перегрузить в классе с помощью функции-члена.

11.3.6. Литералы

Невозможно определить литералы пользовательских типов в том же самом смысле, в каком *1.2* или *12e3* являются литералами типа ***double***. Однако вместо них можно использовать литералы встроенных типов (§4.1.1), если в классе имеются функции-члены для их интерпретации. Конструктор с единственным аргументом служит основным механизмом реализации этого подхода. Когда такие конструкторы просты и поддаются встраиванию, естественно рассматривать вызов конструкторов с литеральными аргументами в качестве литерала. Например, я рассматриваю ***complex(3)*** как литерал типа ***complex***, хотя технически это не так.

11.3.7. Дополнительные функции-члены

До сих пор мы снабдили класс ***complex*** лишь конструкторами и арифметическими операциями. Для практической работы этого не достаточно. В частности, требуются функции, позволяющие читать вещественную и мнимую части комплексного числа:

```
class complex
{
  double re, im;
public:
  double real() const {return re; }
  double imag() const {return im; }
  // ...
};
```

Поскольку ***real***() и ***imag***() не модифицируют значения комплексного числа, то естественно объявить их константными.

Остальные полезные операции можно реализовать, отталкиваясь от ***real***() и ***imag***(), то есть не предоставляя им непосредственного доступа к внутреннему представлению класса ***complex***. Например:

```
inline bool operator==(complex a, complex b)
{
  return a.real()==b.real() && a.imag()==b.imag();
}
```

Часто требуется именно читать вещественную и мнимую части комплексных чисел; перезапись этих значений требуется значительно реже. Если же частичная модификация все же нужна, то можно поступить следующим образом:

```
void f(complex& z, double d)
{
  // ...
  z = complex(z.real(),d);          // присвоить z.im значение d
}
```

Оптимизирующий компилятор может здесь сгенерировать единственную операцию присваивания.

11.3.8. Функции поддержки (helper functions)

Если собрать все фрагменты вместе, то наш класс ***complex*** примет вид:

```
class complex
{
  double re, im;
public:
  complex(double r =0, double i =0) : re(r), im(i) { }

  double real() const {return re; }
  double imag() const {return im; }
```

```
    complex& operator+= (complex) ;
    complex& operator+= (double) ;
    // -=, *=, and /=
};
```

Дополнительно представляем набор глобально определяемых (не в качестве членов класса **complex**) *функций поддержки* (*helper functions*):

```
complex operator+ (complex, complex) ;
complex operator+ (complex, double) ;
complex operator+ (double, complex) ;
// - , *, and /

complex operator- (complex) ;          // унарный минус
complex operator+ (complex) ;          // унарный плюс

bool operator== (complex, complex) ;
bool operator!= (complex, complex) ;

istream& operator>> (istream&, complex&) ; // ввод
ostream& operator<< (ostream&, complex) ; // вывод
```

Отметим, что функции-члены ***real***() и ***imag***() важны для реализации операций сравнения. Они также важны и для иных функций поддержки.

Например, полезно определить функции для работы в полярных координатах:

```
complex polar (double rho, double theta) ;
complex conj (complex) ;

double abs (complex) ;
double arg (complex) ;
double norm (complex) ;

double real (complex) ;                // для удобства записи
double imag (complex) ;                // для удобства записи
```

И, наконец, нужно реализовать подходящий набор стандартных математических функций:

```
complex acos (complex) ;
complex asin (complex) ;
complex atan (complex) ;
// ...
```

С точки зрения пользователя представленный здесь комплексный тип почти идентичен представленному в файле <***complex***> типу ***complex***<***double***> стандартной библиотеки (§22.5).

11.4. Операции приведения типов

Применение конструкторов для приведения типов удобно, но не является панацеей. Конструктор не может задавать:

1. Неявное приведение пользовательского типа во встроенный тип (так как встроенный тип не является классом).

2. Преобразование из нового класса в ранее определенный класс (без модификации старого класса).

Перечисленные проблемы решаются при помощи явно программируемых в классе *операций приведения типов* (*conversion operators*). Функция-член ***X::operator T()***, где ***T*** есть имя типа, определяет приведение типа ***X*** в тип ***T***. Например, пусть определен тип «шестибитовое неотрицательное целое» — ***Tiny***, который можно смешивать с обычными целыми в арифметических операциях:

```
class Tiny
{
  char v;
  void assign (int i)  { if(i&~077) throw Bad_range() ; v=i; }

public:
  class Bad_range{ } ;

  Tiny (int i) { assign (i) ; }
  Tiny& operator= (int i)  {assign (i) ; return*this; }

  operator int() const {return v; }            // операция приведения к типу int
};
```

Диапазон допустимых значений для типа ***Tiny*** проверяется как при его инициализации целыми, так и в присваиваниях ему целых значений. При копировании ***Tiny*** проверка диапазона не нужна, и поэтому мы удовлетворяется умолчательными вариантами копирующего конструктора и операции присваивания.

Чтобы обеспечить возможность обычных вычислений с переменными типа Tiny, мы определяем операцию приведения от ***Tiny*** к ***int***: ***Tiny::operator int()***. Обратите внимание на то, что тип, к которому выполняется преобразование (приведение), присутствует в имени функции-операции и *не может указываться в качестве возвращаемого значения*:

```
Tiny::operator int() const {return v; }        // правильно
int Tiny::operator int() const {return v; }    // ошибка
```

В этом отношении операции приведения типов имеют сходство с конструкторами.

Теперь, если переменные типа ***Tiny*** присутствуют там, где нужны переменные типа ***int***, необходимое преобразование от ***Tiny*** к ***int*** выполняется *неявным образом*:

```
int main()
{
  Tiny c1 = 2;
  Tiny c2 = 62;
  Tiny c3 = c2-c1;    // c3 = 60
  Tiny c4 = c3;       // проверка диапазона не выполняется (нет необходимости)
  int i = c1+c2;      // i = 64

  c1 = c1+c2;         // выход за пределы диапазона: c1 не может равняться 64
  i = c3-64;          // i = -4
  c2 = c3-64;         // выход за пределы диапазона: c2 не может равняться -4
  c3 = c4;            // проверка диапазона не выполняется (нет необходимости)
}
```

Операции приведения особо полезны там, где чтение (реализуемое операцией приведения) тривиально, а инициализация и присваивание существенно менее тривиальны.

Классы ***istream*** и ***ostream*** полагаются на операции приведения, чтобы сделать осмысленными операторы вроде следующего:

```
while (cin>>x) cout<<x;
```

Операция ввода ***cin>>x*** возвращает ***istream&***. Последнее неявно преобразуется к значению, отражающему состояние потока ***cin***. Полученное таким образом значение может использоваться в цикле ***while*** (§21.3.3). Однако в общем случае, идея реализации операций приведения типа, в которых происходит потеря информации, не слишком хороша.

Как правило, лучше не переусердствовать с операциями приведения типов. При избыточном определении возможны неоднозначности. Подобные неоднозначности выявляются компилятором, но от них бывает трудно избавиться. Вероятно, лучше сначала попробовать ограничиться именованными функциями, например ***X::make_int()***. Когда такая функция становится слишком популярной, в результате чего код теряет элегантность, тогда и стоит подумать о введении операции приведения ***X::operator int()***.

Когда же одновременно определяются и пользовательские операции, например +, и пользовательские приведения типа, то могут возникать неоднозначности следующего вида:

```
int operator+ (Tiny, Tiny);

void f(Tiny t, int i)
{
  t+i;           // error, неоднозначность: operator+(t,Tiny(i)) или int(t)+i ?
}
```

Тогда лучше ограничиться чем-либо одним — либо пользовательской операцией сложения, либо пользовательским приведением от ***Tiny*** к ***int***.

11.4.1. Неоднозначности

Присваивание значения типа *V* объекту класса *X* допускается в тех случаях, когда определена операция присваивания ***X::operator=(Z)***, где *V* есть ***Z***, либо имеется уникальное преобразование от *V* к ***Z***. Трактовка инициализации абсолютно аналогична.

В ряде случаев значение требуемого типа может быть создано путем многократного применения конструкторов и операций приведения. Это нужно осуществлять при помощи явных преобразований; *неявное* применение пользовательских преобразований типов осуществляется лишь *однократно*. Иногда значение желаемого типа может быть создано более, чем одним способом — такое допустимо. Например:

```
class X { /*...*/ X(int); X(char*); };
class Y { /*...*/ Y(int); };
class Z { /*...*/ Z(X); };
```

```
X f(X) ;
Y f(Y) ;
Z g(Z) ;

void k1 ()
{
  f(1) ;                // error: неоднозначность: f(X(1)) или f(Y(1))?
  f(X(1)) ;             // ok
  f(Y(1)) ;             // ok
  g("Mack") ;           // error:нужны два пользовательских преобразования;
                        // g(Z(X("Mack"))) not tried
  g(X("Doc")) ;         // ok: g(Z(X("Doc")))
  g(Z("Suzy")) ;        // ok: g(Z(X("Suzy")))
}
```

Пользовательские преобразования типов рассматриваются лишь тогда, когда они необходимы для разрешения неоднозначности вызова. Например:

```
class XX { /*...*/ XX(int) ; } ;

void h (double) ;
void h (XX) ;

void k2 ()
{
  h(1) ;                // h(double(1)) или h(XX(1))? h(double(1))!
}
```

Вызов `h(1)` трактуется как `h(double(1))`, ибо здесь используется стандартное, а не пользовательское преобразование типа (§7.4).

Правила для преобразований и не слишком просты, и весьма сложны в документировании, и не столь общие, как можно было бы подумать. Однако их применение безопасно, а результаты не слишком непредсказуемы. Лучше применить явное разрешение неоднозначности, чем искать ошибку, вызванную неожиданным (неявным) преобразованием.

Требование строгого анализа снизу-вверх предполагает, что тип возвращаемого значения не участвует в разрешении перегрузки. Например:

```
class Quad
{
public:
  Quad(double) ;
  // ...
} ;

Quad operator+ (Quad, Quad) ;

void f(double a1, double a2)
{
  Quad r1 = a1+a2;          // суммирование с двойной точностью
  Quad r2 = Quad(a1)+a2;    // форсируем Quad-арифметику
}
```

Причинами такого подхода являются соображения о том, что анализ снизу вверх более понятен, и что не дело компилятора решать за программиста, какая нужна точность для выполнения сложения.

Когда типы операндов инициализации или присваивания определены, они оба используются для разрешения неоднозначностей. Например:

```
class Real
{
public:
  operator double () ;
  operator int () ;
  // ...
};

void g (Real a)
{
  double d = a;              // d = a.double();
  int i = a;                 // i = a.int();
  d = a;                     // d = a.double();
  i = a;                     // i = a.int();
}
```

И здесь анализ типов идет снизу-вверх: в каждом отдельном случае лишь тип операции и типы операндов используются для разрешения неоднозначности.

11.5. Друзья класса

Обычное объявление функции-члена имеет три логически разных последствия:

1. Функция имеет доступ к закрытой части класса.
2. Функция находится в области видимости класса.
3. Функция вызывается для объекта класса (получает указатель ***this***).

Приписывая функции-члену модификатор ***static*** (§10.2.4), мы обеспечиваем для нее первые два свойства. А если мы припишем функции модификатор ***friend***, то наделим ее одним лишь первым свойством.

Например, можно было бы определить операцию умножения объектов типа ***Matrix*** и ***Vector***. Каждый из этих типов скрывает внутреннее представление и обеспечивает полный набор операций для манипулирования своими объектами. Наша операция умножения не может быть функцией-членом обоих классов, да и вообще, вряд ли мы захотим предоставить любым пользователям низкоуровневые возможности читать/переписывать внутренее устройство объектов типа ***Matrix*** и ***Vector***. Во избежание этого мы объявим операцию умножения другом обоих классов:

```
class Matrix;

class Vector
{
  float v[4] ;
  // ...
  friend Vector operator* (const Matrix&, const Vector&) ;
};
```

```
class Matrix
{
  Vector v[4];
  // ...
  friend Vector operator* (const Matrix&, const Vector&);
};

Vector operator* (const Matrix& m, const Vector& v)
{
  Vector r;
  for (int i = 0; i<4; i++)       // r[i] = m[i] * v;
  {
    r.v[i] = 0;
    for (int j = 0; j<4; j++) r.v[i] += m.v[i].v[j]*v.v[j];
  }
  return r;
}
```

Объявление функции другом можно помещать как в открытые, так и в закрытые секции класса — это *не имеет значения*. Как и функции-члены, функции-друзья явным образом объявляются в классе, друзьями которого они являются. Так что они *являются частью интерфейса класса* в той же мере, что и функции-члены.

Другом класса может быть объявлена и функция-член другого класса. Например:

```
class List_iterator
{
  // ...
  int* next();
};

class List
{
  friend int* List_iterator::next();
  // ...
};
```

Нередко все функции одного класса являются друзьями другого класса. Это можно объявлять короче:

```
class List
{
  friend class List_iterator;
  // ...
};
```

Такое объявление делает все функции класса ***List_iterator*** друзьями класса ***List***.

Ясно, что целые классы друзьями объявляют лишь в случаях, когда имеется теснейшая концептуальная связь между ними. Однако для таких случаев существует и иная альтернатива в виде определения класса *внутри* определения другого класса (*вложенный класс — nested class*) (см. §24.4).

11.5.1. Поиск друзей

Как и объявление функции-члена, объявление друга не вносит его имя в охватывающую область видимости. Например:

```
class Matrix
{
  friend class Xform;
  friend Matrix invert(const Matrix&);
  //...
};

Xform x;                                  // error: Xform нет в текущей области видимости
Matrix (*p) (const Matrix&) = &invert;    // error: invert()нет в текущей области видимости
```

Для больших программ и больших классов это как раз хорошо, что классы не вносят «по-тихому» имена в окружающие области видимости. Это особенно важно для шаблонного класса, который может конкретизироваться в различных контекстах (глава 13).

Дружественный класс должен быть предварительно объявлен в непосредственно охватывающей области видимости, или же определен в той же самой области видимости, где определяется и класс, объявляющий его другом. Более отдаленные (внешние) охватывающие области видимости в расчет не принимаются. Например:

```
class AE{ /*...*/ };          // не друг класса Y

namespace N
{
  class X{ /*...*/ };         // друг класса Y

  class Y
  {
    friend class X;
    friend class Z;
    friend class AE;
  };

  class Z{ /*...*/ };         // друг класса Y
}
```

Дружественная функция может явно объявляться таким же образом, что и дружественный класс, или же ее можно искать по типу аргументов (§8.2.6) даже в случае, когда она объявлена вовсе не в самой близкой из охватывающих областей видимости. Например:

```
void f(Matrix& m)
{
  invert(m);                  // invert() - друг класса Matrix
}
```

Итак, дружественная классу функция или объявляется в непосредственно охватывающей области видимости, или имеет аргумент типа класса (или производного от него класса) (§13.6), или вообще не может быть найдена. Например:

```
// f() нет в этой области видимости

class X
{
 friend void f() ;            // бесполезно
 friend void h (const X&) ;   // можно найти по типу аргумента
};

void g (const X& x) ;
{
 f() ;                        // f() нету в области видимости
 h (x) ;                      // h() - друг класса X
}
```

11.5.2. Функции-члены или друзья?

Когда для реализации некоторой операции нужно использовать функции-члены, а когда дружественные функции? Во-первых, следует свести к минимуму набор функций, имеющих прямой доступ ко внутреннему представлению класса, и определить этот набор самым тщательным образом. Поэтому первым является не вопрос «функция-член или статическая функция или дружественная функция?», а совсем другой вопрос — «требуется ли на самом деле прямой доступ к представлению?». В типичных случаях набор функций-членов с прямым доступом значительно меньше, чем это кажется на первый взгляд.

Конечно, некоторые операции (в смысле, действия) обязаны быть членами — конструкторы, деструктор и виртуальные функции (§12.2.6), но во всех остальных случаях выбор возможен. Так как имена членов класса локальны по отношению к классу, функция, нуждающаяся в прямом доступе к представлению, обязана быть функцией-членом.

Рассмотрим класс *X*, реализующий альтернативные способы определения операций:

```
class X
{
 // ...
 X(int) ;

 int m1 () ;
 int m2 () const;

 friend int f1 (X&) ;
 friend int f2 (const X&) ;
 friend int f3 (X) ;
};
```

Функции-члены можно вызывать только для объектов класса; никаких пользовательских приведений типа не применяется к левым операндам операций . и -> (§11.10). Например:

```
void g ()
{
 99.m1 () ;                   // error: X(99).m1() не рассматривается
 99.m2 () ;                   // error: X(99).m2() не рассматривается
}
```

Глобальная функция *f1*() имеет сходные свойства, так как неявные приведения типа не применяются к неконстантным аргументам, передаваемым по ссылке (§5.5, §11.3.5). Однако к аргументам функций *f2*() и *f3*() преобразования типов применяются:

```
void h ()
{
  f1 (99) ;            // error: f1(X(99)) не рассматривается
  f2 (99) ;            // ok: f2(X(99));
  f3 (99) ;            // ok: f3(X(99));
}
```

Любую функцию, модифицирующую состояние классового объекта, следует определять как функцию-член или глобальную функцию, принимающую объект класса в качестве *неконстантного* аргумента по ссылке (или *неконстантного* указателя). Операции (=, *=, ++ и т.д.), требующие в качестве левого операнда lvalue, для пользовательских типов обычно перегружаются как члены.

И обратно, если для аргументов (операндов) желательно неявное приведение типа, реализующая операцию функция не должна быть членом класса и должна иметь либо аргументы, передаваемые по значению, либо константные аргументы, передаваемые по ссылке. Это типично для функций, реализующих операции, не требующие lvalue в качестве левых операндов (это операции +, -, || и т.д.). В то же время, такие операции часто нуждаются в прямом доступе ко внутреннему представлению класса. Поэтому подобного рода бинарные операции чаще всего являются функциями-друзьями.

Если не определяются пользовательские операции приведения типа, то нет никаких причин отдавать предпочтение функциям-членам перед дружественными функциями со ссылочными аргументами, и наоборот. Часто программисты имеют свои собственные предпочтения. Возникает впечатление, что большинство людей предпочитает вызов *inv*(*m*) для инвертирования матрицы вместо вызова *m*.*inv*(). Однако ж, если *inv*() на самом деле инвертирует объект *m*, а не возвращает новую матрицу, равную инверсии от *m*, то тут предпочтение имеет функция-член.

При прочих равных условиях предпочитайте определять функцию-член. Невозможно предвидеть, не определит ли кто-нибудь когда-нибудь операцию приведения типа. Невозможно предсказать, не потребуют ли будущие модификации изменения состояний используемых объектов. Синтаксис вызова функций-членов делает очевидным для пользователя, что объект может быть изменен; ссылочный аргумент значительно менее очевиден. Кроме того, выражения в теле функции-члена чаще всего короче и проще, чем эквивалентные выражения в глобальных функциях — последним требуются явно задаваемые аргументы, в то время как функции-члены неявно применяют *this*. Наконец, поскольку имена членов локальны в классе, то они могут быть значительно короче, чем имена глобальных функций.

11.6. Объекты больших размеров

Для класса *complex* мы определили типы аргументов как *complex* (эти аргументы передаются по значению, то есть копируются). Накладные расходы на копирование двух *double* имеют определенное значение, но в любом случае это менее накладно, чем применение двух указателей (доступ по указателем относительно дорог). В то

же время, не все классы столь компактны. Для того чтобы избежать накладных расходов на копирование в таких случаях можно задать ссылочные аргументы. Например:

```
class Matrix
{
  double m[4] [4];
public:
  Matrix();
  friend Matrix operator+ (const Matrix&, const Matrix&);
  friend Matrix operator* (const Matrix&, const Matrix&);
};
```

В этом случае использование выражений с перегруженными операциями для объектов большого размера не требует избыточного копирования. Определить операцию сложения можно следующим образом:

```
Matrix operator+ (const Matrix& arg1, const Matrix& arg2)
{
  Matrix sum;

  for (int i=0; i<4; i++)
    for (int j=0; j<4; j++)
      sum.m[i] [j] = arg1.m[i] [j] + arg2.m[i] [j];

  return sum;
}
```

Эта перегруженная операция принимает аргументы (операнды) по ссылке, но возвращает она значение объекта. Возврат по ссылке может показаться более эффективным:

```
class Matrix
{
  // ...
  friend Matrix& operator+ (const Matrix&, const Matrix&);
  friend Matrix& operator* (const Matrix&, const Matrix&);
};
```

Это в принципе допустимо, но вызывает проблемы с выделением памяти. Так как возврат выполняется по ссылке, то возвращаемое значение не может быть автоматическим объектом (§7.3). Так как операция чаще всего используется в выражении не один раз, то ее результат не может быть *статической* локальной переменной. Результат, скорее всего, будет размещаться в свободной памяти. В итоге, копирование возврата обойдется дешевле (с точки зрения времени выполнения, объема памяти и кода), чем выделение памяти под результат и ее последующее освобождение. И программировать это легче.

Существуют приемы, позволяющие избежать копирования результата. Самый простой — использование буфера статических объектов. Например:

```
const max_matrix_temp = 7;

Matrix& get_matrix_temp()
{
  static int nbuf=0;
  static Matrix buf[max_matrix_temp];

  if(nbuf==max_matrix_temp) nbuf=0;

  return buf[nbuf++];
}

Matrix& operator+ (const Matrix& arg1, const Matrix& arg2)
{
  Matrix& res = get_matrix_temp();
  //...
  return res;
}
```

Теперь объект типа ***Matrix*** копируется только тогда, когда результат вычисления выражения присваивается чему-либо. Однако лишь небеса могут помочь вам в том случае, если потребуются более чем ***max_matrix_temp*** временных объектов.

Менее чреватая ошибками техника определяет матрицу как *дескриптор* (*handle*; §25.7) к *классу-представлению* (*representation type*), содержащему действительные данные. В этом случае, матричные дескрипторы могут так управлять объектами представления, что затраты на выделение памяти и копирование будут минимальными (§11.12 и §11.14[18]). Эта техника базируется на возврате объектов, а не ссылок или указателей. Еще одна техника опирается на определение тернарных операций и их автоматический вызов для выражений вида ***a=b+c*** или ***a+b*i*** (§21.4.6.3, §22.4.7).

11.7. Важные операции

В общем случае, для типа ***X*** копирующий конструктор ***X(const X&)*** выполняет инициализацию посредством объекта того же самого класса ***X***. Важно еще раз напомнить, что инициализация и присваивание — это разные операции (§10.4.4.1). Особенно это важно для классов с деструктором. Когда класс имеет нетривиальный деструктор, освобождающий, например, ранее выделенную конструктором память, скорее всего этот класс нуждается в полном наборе средств, управляющих конструированием объектов, их уничтожением и копированием:

```
class X
{
  //...
  X(Sometype);                // конструктор
  X(const X&);                // копирующий конструктор
  X& operator= (const X&);    // операция присваивания: очистка и копирование
  ~X();                       // деструктор: очистка
};
```

Объект копируется еще в трех случаях: при передаче аргумента функции, при возврате из функции, и при генерации исключений. При передаче аргумента ини-

циализируется формальный параметр. Семантика здесь полностью аналогична таковой для любой инициализации. То же самое имеет место и для возвращаемых из функций значений, и для исключений, хотя и в менее очевидной форме. Во всех этих случаях применяется копирующий конструктор. Например:

```
string g (string arg)  // передача по значению (используется копирующий конструктор)
{
  return arg;          // возвращается строка  (используется копирующий конструктор)
}

int main ()
{
  string s= "Newton";
  s = g (s) ;
}
```

Очевидно, что строка *s* будет содержать "*Newton*" после вызова функции *g*(). Передача копии *s* в аргумент *arg* не составляет труда — вызов конструктора класса *string* поможет это выполнить. Получение еще одной копии при выходе из *g*() требует еще одного вызова *string*(*const string*&) ; на этот раз инициализируется временная переменная (§10.4.10), которая затем присваивается переменной *s*. Часто одна из этих операций (но не обе сразу) может устраняться в процессе оптимизации.

Для класса, в котором копирующий конструктор и операция присваивания не определяются явным образом, предоставляются соответствующие умолчательные варианты от компилятора (§10.2.5). Это подразумевает, что копирующие операции не наследуются (§12.2.3).

11.7.1. Конструктор с модификатором explicit

Как мы уже знаем, конструктор с одним аргументом определяет неявное приведение типа. Для некоторых типов это является идеальным решением. Например, тип *complex* может инициализироваться типом *int*:

```
complex z = 2;          // инициализация z значением complex(2)
```

В других случаях неявное приведение типа нежелательно и может приводить к ошибкам. Например, если бы мы могли инициализировать *string* размером (имеет тип *int*), кто-нибудь написал бы:

```
string s = 'a' ;        // создается строка s с int('a') элементами
```

Это прошло бы, но вряд ли соответствовало бы настоящему намерению программиста.

Неявные приведения типа можно подавить, объявив конструктор с ключевым словом *explicit*. Такой конструктор можно вызывать только явно. В частности, там, где принципиально требуется копирующий конструктор (§11.3.4), конструктор с модификатором *explicit* вызываться неявно уже не будет. Например:

```
class String
{
  // ...
  explicit String (int n) ;       // выделить n байт
  String (const char* p) ;        // инициализировать С-строкой
};
```

```
String s1 = 'a';              // error: неявного преобразования char–>String нет
String s2(10);                // ok: String с местом для 10 символов
String s3 = String(10);       // ok: String с местом для 10 символов
String s4 = "Brian";          // ok: s4 = String("Brian")
String s5("Faulty");

void f(String);

String g()
{
 f(10);                       // error: неявного преобразования int->String нет
 f(String(10));
 f("Arthur");                 // ok: f(String("Arthur"))
 f(s1);

 String* p1 = new String("Eric");
 String* p2 = new String(10);

 return 10;                   // error: неявного преобразования int->String нет
}
```

Может быть разница между

```
String s1 = 'a';              // error: неявного преобразования char–>String нет
```

и

```
String s2(10);                // ok: String с местом для 10 символов
```

покажется слишком тонкой, но она более заметна в реальном коде, чем в надуманных примерах.

В классе ***Date*** мы использовали тип ***int*** для представления года (§10.3). Будь наше отношение к разработке ***Date*** более серьезным, мы могли бы создать класс ***Year*** для обеспечения более строгой проверки во время компиляции. Например:

```
class Year
{
 int y;

public:
 explicit Year(int i) : y(i) {}       // создание Year из int
 operator int() const {return y;}     // преобразование: Year в int
};

class Date
{
public:
 Date(int d, Month m, Year y);
 // ...
};

Date d3(1978,feb,21);                 // error: 21 это не Year
Date d4(21,feb,Year(1978));           // ok
```

Класс ***Year*** является простой «оберткой» («wrapper») вокруг типа ***int***. Благодаря наличию определения для ***operator int()*** тип ***Year*** неявно преобразуется в ***int*** каждый раз, когда это необходимо. Из-за того, что мы снабдили конструктор модифи-

каторном ***explicit***, мы можем быть уверены, что преобразования из ***int*** в ***Year*** происходят только тогда, когда мы об этом просим *явно*, а случайные оказии вылавливаются компилятором. Функции-члены класса ***Year*** легко превращаются во встраиваемые функции, так что нет никаких дополнительных затрат по памяти и быстродействию.

Аналогичную технику можно применить и к диапазонам (§25.6.1).

11.8. Индексирование

Функцию-операцию ***operator***[] определяют с целью придать смысл операциям индексации над объектами классов. Второй аргумент этой операции (индекс) может иметь любой тип. Это и позволяет определять такие типы стандартной библиотеки, как вектора, ассоциативные массивы и т.д.

Для иллюстрации перепишем пример из §5.5, в котором ассоциативный массив использовался для подсчета количества вхождений слова в содержимое файла. Там для этого использовалась функция. Сейчас же мы определим свой *собственный ассоциативный массив*:

```
class Assoc
{
  struct Pair
  {
    string name;
    double val;
    Pair (string n= " " ,double v =0)  : name(n), val(v) {}
  };

  vector<Pair> vec;

  Assoc(const Assoc&);                  // private - для предотвращения копирования
  Assoc& operator= (const Assoc&);      // private - для предотвращения копирования

public:
  Assoc() {}
  const double& operator[] (const string&);
  double& operator[] (string&);
  void print_all() const;
};
```

Класс ***Assoc*** хранит вектор элементов типа ***Pair***. Реализация использует тот же самый тривиальный и неэффективный метод поиска, что и в §5.5:

```
// поиск s; возврат ее значения в случае нахождения; в противном случае
// создаем новый Pair и возвращаем умолчательное значение 0
//
double& Assoc::operator[] (string& s)
{
  for (vector<Pair>::iterator p = vec.begin();p!=vec.end(); ++p)
    if(s==p->name) return p->val;

  vec.push_back(Pair(s,0));         // начальное значение: 0
  return vec.back().val;            // возврат последнего элемента (§16.3.3)
}
```

Так как класс *Assoc* скрывает свое внутреннее представление, то требуется открытый метод для вывода его содержимого:

```
void Assoc::print_all() const
{
  for (vector<Pair>::const_iterator p = vec.begin();p!=vec.end();++p)
    cout<< p->name << " : " << p->val << '\n';
}
```

Вот пример простого клиентского кода, использующего класс *Assoc*:

```
int main()   // подсчитать кол-во вхождений каждого слова во входном потоке
{
  string buf;
  Assoc vec;

  while(cin>>buf) vec[buf]++;

  vec.print_all();
}
```

Дальнейшее развитие идеи ассоциативного массива представлено в §17.4.1.

Операцию `[]` можно перегружать *только* в виде *функции-члена.*

11.9. Функциональный вызов

Вызов функции, то есть выражение вида *expression*(*expression-list*), интерпретируется как бинарная операция с левым операндом *expression* (имя функции или именующее выражение) и правым операндом *expression-list* (список аргументов). Эту операцию можно перегружать, как и большинство других операций C++. При этом список аргументов для функции-операции **operator**`()` `()` вычисляется и проверяется по обычным правилам передачи аргументов. Чаще всего перегрузку операции вызова функции осуществляют для классов, имеющих единственную или доминирующую над другими операцию.

Самым очевидным и, вероятно, наиболее полезным применением перегруженной операции `()` является предоставление обычного синтаксиса функциональных вызовов классовым объектам, которые в некотором отношении ведут себя как функции. Объекты, которые ведут себя как функции, принято называть функциональными объектами или *объектами-функциями* (*function objects*) (§18.4)[1]. Объекты-функции важны, так как позволяют писать код, принимающий нетривиальные операции в качестве параметров. Например, в стандартной библиотеке реализовано множество алгоритмов, которые вызывают функцию для каждого элемента контейнера. Рассмотрим пример:

```
void negate(complex& c) {c = -c;}

void f(vector<complex>& aa, list<complex>& ll)
{
  for_each(aa.begin(),aa.end(),negate);    // поменять знак у всех эл-ов вектора
  for_each(ll.begin(),ll.end(),negate);    // поменять знак у всех эл-ов списка
}
```

[1] Также их часто называют *функторами* (*functors*). — *Прим. ред.*

Этот код предназначен для изменения знака (на противоположный) у каждого элемента вектора и списка.

А что, если нам потребуется добавить *complex(2,3)* к каждому элементу? Это легко можно сделать следующим образом:

```
void add23(complex& c)
{
  c += complex(2,3);
}

void g(vector<complex>& aa, list<complex>& ll)
{
  for_each(aa.begin(),aa.end(),add23);
  for_each(ll.begin(),ll.end(),add23);
}
```

А как написать функцию, прибавляющую к каждому элементу произвольное комплексное значение? В этом случае нам потребуется нечто, чему мы можем передать произвольное значение, и что затем будет использовать это значение при каждом вызове. Это невозможно проделать с функциями, оставаясь лишь в контексте функций. Требуется передавать наше значение в некоторый констекст, окружающий функцию. Все это довольно запутанно. В то же время, нетрудно написать класс, демонстрирующий указанное поведение:

```
class Add
{
  complex val;

public:
  Add(complex c) {val=c;}                       // сохраняем значение
  Add(double r, double i) {val=complex(r,i);}

  void operator()(complex& c) const {c+=val;}   // прибавляем значение к аргументу
};
```

Объект класса *Add* инициализируются произвольно задаваемым комплексным числом, а когда затем к этому объекту обращаются с помощью операции (), он прибавляет комплексное число к аргументу. Например:

```
void h(vector<complex>& aa, list<complex>& ll, complex z)
{
  for_each(aa.begin(),aa.end(),Add(2,3));
  for_each(ll.begin(),ll.end(),Add(z));
}
```

Здесь комплексное значение *complex(2,3)* будет добавлено к каждому элементу вектора, и комплексное число *z* будет добавлено к каждому элементу списка. Заметьте, что с помощью выражения *Add(z)* конструируется объект класса *Add*, который затем многократно используется алгоритмом *for_each()*; *Add(z)* не является функцией, вызываемой однажды или повторно. Повторно вызывается функция-член *operator()()* класса *Add*.

Все это работает потому еще, что *for_each* является функциональным шаблоном, применяющим операцию () к своему третьему аргументу, даже не заботясь о том, чем он является в действительности:

```
template<class Iter, class Fct> Fct for_each (Iter b, Iter e, Fct f)
{
  while (b! = e) f(*b++) ;
  return f;
}
```

На первый взгляд может показаться, что такая техника является чем-то эзотерическим, но на самом деле все это не так уж и сложно, весьма эффективно и чрезвычайно полезно (см. §3.8.5, §18.4).

Другими популярными задачами для применения **operator** () () являются задача выделения подстроки и задача индексирования многомерных массивов (§22.4.5).

Функция **operator** () () может быть *только функцией-членом.*

11.10. Разыменование

Операция разыменования -> *перегружается в классах как унарная и постфиксная*:

```
class Ptr
{
  //...
  X* operator-> () ;
};
```

Объектами класса **Ptr** можно пользоваться для доступа к членам класса *X* точно так же, как обычными указателями. Например:

```
void f(Ptr p)
{
  p->m = 7;      // (p.operator->())->m = 7
}
```

Переход от объекта *p* к возвращаемому вызовом *p*.**operator->** () указателю не зависит от указуемого члена m. В этом смысле операция разыменования ->, перегружаемая с помощью функции **operator->** (), является *унарной* (и *постфиксной*). При этом никакого нового синтаксиса не вводится, так что по-прежнему после знака операции нужно указать имя адресуемого операцией члена. Например:

```
void g (Ptr p)
{
  X* q1 = p->;                  // синтаксическая ошибка
  X* q2 = p.operator-> () ;     // ok
}
```

Перегрузка операции -> используется преимущественно для построения так называемых *интеллектуальных указателей* (*smart pointers*), то есть классовых объектов, которые ведут себя как указатели и дополнительно выполняют некоторые действия, когда через них осуществляется доступ к целевому объекту. Например, можно определить класс **Rec_ptr** для доступа к объектам класса **Rec**, хранящимся на диске. Конструктор класса **Rec_ptr** принимает имя, которое используется для доступа к объекту на диске, **Rec_ptr::operator->**() извлекает объект с диска

в оперативную память, а деструктор записывает измененное значение объекта на диск:

```
class Rec_ptr
{
  const char* identifier;
  Rec* in_core_address;
  // ...
public:
  Rec_ptr(const char* p) : identifier(p), in_core_address(0) {}
  ~Rec_ptr() { write_to_disk(in_core_adress,identifier); }

  Rec* operator->();
};

Rec* Rec_ptr::operator->()
{
  if(in_core_address==0) in_core_address=read_from_disk(identifier);
  return in_core_address;
}
```

Rec_ptr можно использовать следующим образом:

```
struct Rec                 // Rec, на который указывает Rec_ptr
{
  string name;
  // ...
};

void update(const char* s)
{
  Rec_ptr p(s);            // Rec_ptr для s
  p->name="Roscoe";        // обновить s (при необходимости считать с диска)
  // ...
}
```

В реальной жизни ***Rec_ptr*** был бы шаблоном, а тип ***Rec*** был бы его параметром. Кроме того, реальная программа обязана обрабатывать ошибочные ситуации, а ее взаимодействие с диском было бы не столь наивным, как в нашем учебном примере.

Для обычных указателей языка C++ применение операции -> синонимично комбинации операций . и *. Пусть для типа ***Y*** операции ->, ., * и [] имеют стандартный смысл, и указатель ***p*** имеет тип ***Y****. Тогда справедливы следующие равенства:

```
p->m==(*p).m          // истина
(*p).m==p[0].m        // истина
p->m==p[0].m          // истина
```

Как обычно, для пользовательских операций таких автоматических гарантий нет. Необходимую эквивалентность операций нужно программировать самому:

```
class Ptr_to_Y
{
  Y* p;
```

```
public:
  Y* operator-> () {return p; }
  Y& operator* () {return *p; }
  Y& operator[] (int i) {return p[i]; }
};
```

Действительно, если вы предоставляете для пользовательского типа более одной подобной операции, будет разумно обеспечить ожидаемую пользователями эквивалентность точно так же, как бывает полезно обеспечить эквивалентность ++*x*, *x*+=*1* и *x*=*x*+*1* для объекта *x* класса, перегружающего операции ++, +=, = и +.

Перегрузка операции -> представляет интерес для целого круга задач. Дело в том, что *косвенные обращения* (*indirection*) составляют важную теоретическую концепцию, а в программировании на C++ операция -> реализует эту концепцию самым прямым и эффективным образом. Итераторы (глава 19) служат еще одним примером реализации концепции косвенных обращений. Еще один взгляд на перегрузку операции -> состоит в том, что с ее помощью реализуется одна из форм концепции *делегирования* (*delegation*; §24.3.6).

Операция -> может перегружаться *только в виде функции-члена*. Типом *возвращаемого* ею *значения* может быть либо *указатель*, либо *объект класса, перегружающего операцию* ->.

11.11. Инкремент и декремент

Если определены «интеллектуальные указатели», то стоит дополнить их перегрузкой сопутствующих традиционным указателям операций ++ (инкремент) и -- (декремент). Особенно это важно в случаях, когда интеллектуальные указатели предназначаются для замены указателей встроенных. К примеру, рассмотрим следующую традиционную программу со свойственными ей недостатками:

```
void f1 (T a)
{
  T v[200] ;
  T* p = &v[0] ;

  p--;
  *p=a;          // Oops: p вне корректного диапазона (и это не перехватывается)

  ++p;
  *p= ;          // ok
}
```

Для преодоления проблем с традиционными указателями мы могли бы заменить указатель *p* объектом класса *Ptr_to_T*, разыменовывать который можно лишь в тех случаях, когда он на самом деле указывает на целевой объект. Также было бы неплохо, если операции инкремента и декремента можно было применять к нему только тогда, когда указуемый объект находится в составе массива объектов. То есть нам требуется что-то вроде следующего:

```
class Ptr_to_T
{
  //...
};
```

```
void f2 (T a)
{
  T v[200] ;
  Ptr_to_T p (&v[0] , v, 200) ;

  p--;
  *p c = a;          // ошибка времени выполнения (run-time error): p вне диапазона

  ++p;
  *p = a;            // ok
}
```

Операции инкремента и декремента уникальны в том отношении, что они могут применяться префиксно и постфиксно. Отсюда следует, что мы должны перегрузить оба варианта для класса ***Ptr_to_T***. Например:

```
class Ptr_to_T
{
  T* p;
  T* array;
  int size;

public:
  Ptr_to_T (T* p, T* v, int s) ;   // привязка к массиву v размера s; нач. значение p
  Ptr_to_T (T* p) ;                // привязка к объекту; начальное значение p

  Ptr_to_T& operator++ () ;        // префиксный
  Ptr_to_T operator++ (int) ;      // постфиксный

  Ptr_to_T& operator-- () ;        // префиксный
  Ptr_to_T operator-- (int) ;      // постфиксный

  T& operator* () ;                // префиксная операция
};
```

Аргумент типа ***int*** используется для указания на то, что функция должна вызываться для постфиксной формы операции инкремента/декремента. Причем *величина* аргумента *не играет никакой роли*; этот аргумент служит лишь тупым указанием на форму операции. Легко запомнить, для какой именно формы операций инкремента/декремента не нужен фиктивный аргумент — он не нужен для обычной префиксной формы, как он не нужен и для остальных обычных унарных арифметических и логических операций, а нужен он лишь для «странных» постфиксных ++ и --.

Приведем пример использования последнего варианта класса ***Ptr_to_T***:

```
void f3 (T a)
{
  T v[200] ;
  Ptr_to_T p (&v[0] , v, 200) ;

  P operator-- (0) ;
  p.operator* () =a;   // ошибка времени выполнения (run-time error): p вне диапазона

  p.operator++ () ;
  p.operator* () =a;   // ok
}
```

Завершение класса *Ptr_to_T* оставлено в качестве самостоятельного упражнения (§11.14[19]). Его дальнейшее превращение в шаблон с генерацией исключений для сообщений об ошибках составляет задачу еще одного упражнения (§14.12[2]). Работа операций ++ и -- над итераторами рассматривается в §19.3. «Шаблонные указатели», корректно ведущие себя по отношению к наследованию, рассматриваются в §13.6.3.

11.12. Класс строк

В данном разделе представлена достаточно реалистичная версия строкового класса *String*. Это минимальная версия, которая меня устраивает. Она обеспечивает семантику значений, операции посимвольного чтения и записи, контролируемый и неконтролируемый доступ, потоковый ввод/вывод, операции сравнения и конкатенации. Класс хранит строку в виде массива символов с терминальным нулем и использует *счетчик ссылок* (*reference count*) для минимизации копирования. Дальнейшее усовершенствование этого класса является полезным упражнением (§11.14[7–12]). А затем мы можем выбросить все наши упражнения и использовать класс строк из стандартной библиотеки (глава 20).

Мой почти настоящий класс *String* опирается на три вспомогательных класса: класс *Srep*, который позволяет разделять истинное представление между несколькими объектами типа *String*, имеющими одинаковое значение; класс *Range*, объекты которого генерируются в виде исключений с сообщениями о выходе из допустимого диапазона; класс *Cref* — помогающий в реализации операции индексирования, различающей чтение и запись:

```
class String
{
  struct Srep;        // представление (representation)
  Srep* rep;
  class Cref;         // ссылка на char

public:
  class Range { };    // для исключений
  // ...
};
```

Как и остальные члены класса, *вложенный класс* (*nested class*) может быть внутри объемлющего класса лишь объявлен, а определен позже:

```
struct String::Srep
{
  char* s;            // указатель на элементы
  int sz;             // кол-во символов
  int n;              // счетчик ссылок

  Srep (int nsz, const char* p)
  {
    n = 1;
    sz = nsz;
    s = new char[sz+1 ];        // плюс терминальный нуль
```

```
    strcpy(s,p);
  }

  ~Srep() { delete[] s; }

  Srep* get_own_copy()          // клонировать при необходимости
  {
    if(n==1) return this;
    n--;
    return new Srep(sz,s);
  }

  void assign(int nsz, const char* p)
  {
    if(sz!=nsz)
    {
      delete[] s;
      sz = nsz;
      s = new char[sz+1];
    }
    strcpy(s,p);
  }
private:                        // предотвращаем копирование:
  Srep(const Srep&);
  Srep& operator=(const Srep&);
};
```

Класс *String* обеспечивает обычный набор конструкторов, деструктор и операции присваивания:

```
class String
{
  // ...
  String();                     // x = ""
  String(const char*);          // x = "abc"
  String(const String&);        // x = other_string
  String& operator=(const char*);
  String& operator=(const String&);
  ~String();
  // ...
};
```

Класс *String* реализует семантику значений. То есть после присваивания *s1=s2* строки *s1* и *s2* абсолютно различимы и последующие изменения одной из них никак не затрагивают другую строку. Альтернативой является семантика указателей, при которой после *s1=s2* изменение *s1* влекло бы за собой и изменение *s2*. Для типов с обычными арифметическими операциями, таких как комплексные числа, вектора, матрицы и строки, я предпочитаю семантику значений. Однако чтобы обеспечить семантику значений, класс *String* реализован в виде *дескриптора* (*handle*) к *классу представления* (*representation class*), которое копируется лишь при необходимости:

```
String::String()
{
```

```
    rep = new Srep(0,"");              // умолчательное значение - пустая строка
}

String::String(const String& x)
{
    x.rep->n++;
    rep = x.rep;                       // разделяемое представление
}
String::~String()
{
    if(--rep->n==0) delete rep;
}

String& String::operator=(const String& x)
{
    x.rep->n++;                        // защита от "st=st"
    if(--rep->n==0) delete rep;
    rep = x.rep;                       // разделяемое представление
    return *this;
}
```

Операции псевдокопирования с аргументами типа ***const char**** позволяют работать со строковыми литералами:

```
String::String(const char* s)
{
    rep = new Srep(strlen(s),s);
}

String& String::operator=(const char* s)
{
    if(rep->n==1)
        rep->assign(strlen(s),s);      // используем старый Srep
    else
    {
        rep->n--;
        rep = new Srep(strlen(s),s);   // используем новый Srep
    }
    return *this;
}
```

Проектирование операций доступа к строкам является непростой материей, поскольку в идеале хотелось бы доступа в привычной форме (операцией `[]`), максимально эффективного и с проверкой диапазона индексов. К сожалению, невозможно обеспечить все эти свойства одновременно. Мой выбор таков: эффективные операции без проверки диапазона со слегка неудобной формой записи плюс менее эффективные операции с проверкой диапазона и со стандартной формой записи:

```
class String
{
    // ...

    void check(int i) const {if(i<0 || rep->sz<=i) throw Rangee();}
```

```
    char read (int i) const {return rep->s[i];}
    void write (int i, char c) {rep=rep->get_own_copy(); rep->s[i]=c;}

    Cref operator[] (int i) {check (i); return Cref(*this,i);}
    char operator[] (int i) const {check (i); return rep->s[i];}

    int size() const {return rep->sz;}
    // ...
};
```

Идея состоит в том, чтобы использовать операцию `[]` с проверкой диапазона в общем случае, но предоставить пользователю оптимизированную возможность однократной проверки диапазона для серии обращений. Например:

```
int hash (const String& s)
{
    int h = s.read(0);
    const int max = s.size();

    for (int i = 1; i<max; i++) h ^= s.read(i)>>1;    // обращение к s без проверки
    return h;
}
```

Определять работу операции `[]` для чтения и записи сложно в тех случаях, когда не допустимо просто вернуть ссылку и позволить делать пользователю, что ему вздумается. В нашем случае такое решение не подходит, поскольку я определил класс ***String*** таким образом, что между несколькими экземплярами класса ***String***, которые присваивались, передавались в качестве аргументов и т.д., представление разделяется до тех пор, пока не потребуется фактическая запись. Тогда, и только тогда, представление копируется. Этот прием называют *копированием по фактической записи* (*copy-on-write*). Само копирование выполняется функцией ***String::Srep::get_own_copy()***.

Чтобы можно было функции доступа сделать встраиваемыми, нужно помещать их определения в области видимости ***Srep***, что в свою очередь можно обеспечить либо помещением определения ***Srep*** внутри определения ***String***, либо определением функций доступа с модификатором ***inline*** вне ***String*** и после ***String::Srep*** (§11.14[2]).

Чтобы различать чтение и запись, ***String::operator[]()*** возвращает ***Cref***, если он вызван для неконстантного обьекта. Тип ***Cref*** ведет себя как ***char&***, за исключением того, что он вызывает ***String::write()*** для операции присваивания:

```
class String::Cref        // ссылка на s[i]
{
friend class String;

    String& s;
    int i;

    Cref(String& ss, int ii) : s(ss), i(ii) { }
    Cref(const Cref& r) : s(r.s), i(r.i) { }
    Cref();                    // не определяется (не используется)

public:
    operator char() const { s.check(i); return s.read(i); }     // выдает значение
```

```
  void operator= (char c) { s.write(i, c); }        // изменяет значение
};
```

Например:

```
void f(String s, const String& r)
{
  char c1=s[1];     // c1=s.operator[](1).operator char()
  s[1] = 'c';       // s.operator[](1).operator=('c')

  char c2 = r[1];   // c2 = r.operator[](1)
  r[1] = 'd';       // error: присваивание для non-lvalue char, r.operator[](1) = 'd'
}
```

Обратите внимание на то, что для неконстантного объекта ***s.operator*[] (*1*)** есть ***Cref*(*s*, *1*)**.

Для полноты я снабдил класс ***String*** набором полезных функций:

```
class String
{
  // ...
  String& operator+= (const String&);
  String& operator+= (const char*);

  friend ostream& operator<< (ostream&, const String&);
  friend istream& operator>> (istream&, String&);

  friend bool operator== (const String& x, const char* s)
   { return strcmp(x.rep->s, s) == 0; }

  friend bool operator== (const String& x, const String& y)
   { return strcmp(x.rep->s, y.rep->s) == 0; }

  friend bool operator!= (const String& x, const char* s)
   { return strcmp(x.rep->s, s) != 0; }

  friend bool operator!= (const String& x, const String& y)
   { return strcmp(x.rep->s, y.rep->s) != 0; }
};

String operator+ (const String&, const String&);
String operator+ (const String&, const char*);
```

Операции ввода/вывода я оставляю в качестве упражнения.

Вот пример простого клиентского кода, использующего операции класса ***String***.

```
String f(String a, String b)
{
  a[2] = 'x';
  char c = b[3];
  cout<< "in f: "<< a << ' ' << b << ' ' << c << '\n';
  return b;
}

int main()
{
  String x, y;
```

```
    cout<<"Please enter two strings\n";
    cin >> x >> y;
    cout << "input: " << x << ' ' << y << '\n';

    String z = x;
    y = f(x,y);
    if(x != z) cout<< "x corrupted!\n";

    x[0] = '!';
    if(x == z) cout<< "write failed!\n";
    cout<<< "exit: " << x << ' ' << y << ' ' << z << '\n';
}
```

В классе ***String*** вы можете обнаружить отсутствие многих возможностей, которые считаете важными или даже необходимыми. Например, вам может недоставать возможности представления содержимого в виде С-строки (§11.14[10], глава 20).

11.13. Советы

1. Перегружайте операции для имитации традиционных форм записи; §11.1.
2. Операнды больших размеров передавайте как *константные* ссылки; §11.6.
3. Для возвратов большого размера рассмотрите возможность оптимизации; §11.6.
4. Предпочитайте умолчательное копирование от компилятора, если оно подходит для вашего класса; §11.3.4.
5. Переопределите или запретите умолчательное копирование от компилятора, если оно не подходит для вашего класса; §11.2.2.
6. Предпочитайте функции-члены, если требуется прямой доступ к представлению класса; §11.5.2.
7. Предпочитайте функциям-членам глобальные функции, если не требуется прямой доступ к представлению класса; §11.5.2.
8. Используйте пространства имен для связывания класса с функциями поддержки; §11.2.4.
9. Используйте глобальные функции для реализации симметричных операций; §11.3.2.
10. Для индексации многомерных массивов используйте операцию (); §11.9.
11. Снабжайте конструктор с одним аргументом для задания размера модификатором ***explicit***; §11.7.1.
12. В обычных случаях предпочитайте стандартный класс ***string*** (глава 20) своим собственным разработкам; §11.12.
13. Будьте осторожны, определяя неявные приведения типа; §11.4.
14. Для реализации операций, требующих lvalue в качестве левого операнда, используйте функции-члены; §11.3.5.

11.14. Упражнения

1. (*2) В следующей программе, какие преобразования выполняются в каждом выражении?

```
struct X
{
  int i;
  X(int) ;
  X operator+ (int) ;
};

struct Y
{
  int i;
  Y(X) ;
  Y operator+ (X) ;
  operator int() ;
};

extern X operator* (X, Y) ;
extern int f(X) ;

X x = 1;
Y y = x;
int i = 2;

int main ()
{
  i+10;      y+10;      y+10*y;
  x+y+i;     x*x+i;     f(7) ;
  f(y) ;     y+y;       106+y;
}
```

   Модифицируйте программу так, чтобы при запуске она выводила значения допустимых выражений.

2. (*2) Завершите и оттестируйте класс ***String*** из §11.12.
3. (*2) Определите класс ***INT***, который ведет себя в точности как ***int***. Подсказка: определите ***INT::operator int()***.
4. (*1) Определите класс ***RINT***, который ведет себя как ***int***, за исключением того, что допустимыми операциями являются лишь +, – (унарные и бинарные), *, / и %. Подсказка: не определяйте ***RINT::operator int()***.
5. (*3) Определите класс ***LINT***, который ведет себя как ***RINT***, но для представления чисел использует по крайней мере 64 бита.
6. (*4) Определите класс с арифметикой произвольной точности. Протестируйте этот класс, вычислив факториал от ***1000***. Подсказка: вам потребуется управление памятью вроде того, что было сделано для класса ***String***.
7. (*2) Для класса ***String*** определите внешний итератор:

```
class String_iter          // ссылается на строку и элемент строки
{
public:
```

```
String_iter (String& s) ;   // итератор для s
char& next () ;             // ссылка на следующий элемент
// иные операции по вашему выбору
};
```

Сравните это по удобству, стилю программирования и эффективности с внутренним итератором для ***String*** (в смысле понятия текущего элемента ***String*** и операций над этим элементом).

8. (*1.5) Для строкового класса определите операцию выделения подстроки с помощью `()`. Какие еще операции вам нужны для работы со строками?
9. (*3) Разработайте класс ***String*** таким образом, чтобы операцию взятия подстроки можно было использовать в левой части операции присваивания. Сначала ограничьтесь версией, где длины подстроки и присваиваемой строки одинаковые. Затем напишите версию с разными длинами.
10. (*2) Напишите операцию для класса ***String***, позволяющую получить содержимое в виде С-строки. Обсудите все за и против способа реализации этой операции в виде операции приведения типа. Обсудите возможные альтернативы выделения памяти под С-строку.
11. (*2.5) Реализуйте простой механизм регулярных выражений для поиска в строках типа ***String***.
12. (*1.5) Модифицируйте механизм поиска из §11.14[11] для работы со стандартным классом ***string***. Заметьте, что вы не можете модифицировать класс ***string***.
13. (*2) Напишите программу, которая совершенно нечитабельна из-за применения макросов и определения пользовательских операций. Идея: определите операцию + так, чтобы она означала - (и наоборот) для ***INT***, а затем макросом определите ***INT*** как ***int***. Переопределите популярные функции, чтобы они принимали ссылочные аргументы. Ряд запутывающих комментариев окончательно «украсят» ваше произведение.
14. (*3) Передайте результат §11.14[13] вашему другу. Посмотрите за тем, как ваш друг будет разбираться со смыслом программы. В конце концов, вы поймете, чего вам следует избегать.
15. (*2) Определите тип ***Vec4*** как вектор из четырех элементов типа ***float***. Определите операцию `[]` для ***Vec4***. Определите операции +, -, *, /, =, +=, -=, *= и /= для комбинации операндов типа ***Vec4*** и типа ***float***.
16. (*3) Определите класс ***Mat4*** как вектор четырех элементов типа ***Vec4***. Определите операцию `[]` для типа ***Mat4*** так, чтобы она возвращала ***Vec4***. Определите обычные матричные операции для ***Mat4***. Определите функцию, реализующую метод исключений Гаусса.
17. (*2) Определите класс ***Vector***, аналогичный ***Vec4***, но с размером, задаваемым конструктором `Vector::Vector(int)`.
18. (*3) Определите класс ***Matrix***, аналогичный ***Mat4***, но с размерами, задаваемыми конструктором `Matrix::Matrix(int, int)`.

19. (*2) Завершите класс ***Ptr_to_T*** из §11.11 и протестируйте его. По крайней мере определите для этого класса операции *, ->, =, ++ и --. Сделайте так, чтобы ошибка времени выполнения возникала только при попытке разыменования недействительных ссылок.

20. (*1) При наличии определений

```
struct S {int x, y; };
struct T {char* p; char* q; };
```

дайте определение класса С, позволяющего использовать ***x*** и ***p*** из ***S*** и ***T*** примерно так же, как если бы они были членами С.

21. (*1.5) Определите класс ***Index*** для хранения индекса функции возведения в степень ***mypow(double, Index)***. Найдите возможность для ***2**I*** отрабатывать в виде вызова ***mypow(2,I)***.

22. (*2) Определите класс ***Imaginary*** для представления мнимых чисел. Определите класс ***Complex*** на базе класса ***Imaginary***. Реализуйте основные арифметические операции.

12

Наследование классов

Не множьте объекты без необходимости.
— В. Оккам

Концепции и классы — производные классы — функции-члены — конструирование и уничтожение — иерархии классов — поля типа — виртуальные функции — абстрактные классы — традиционные иерархии классов — абстрактные классы как интерфейсы — локализация создания объекта — абстрактные классы и иерархии классов — советы — упражнения.

12.1. Введение

Из языка Simula язык C++ позаимствовал концепцию класса как пользовательского типа и концепцию классовых иерархий. Дополнительно он заимствовал фундаментальную идею о том, что классы моделируют концепции из мира программ и из реального мира. Язык C++ содержит конструкции, которые непосредственно поддерживают концепции проектирования. Применение таких языковых конструкций с целью прямого воплощения дизайнерских идей свидетельствует об эффективном использовании C++. В то же время, если эти конструкции используются в качестве необязательных подпорок для традиционных стилей программирования, то это не самый эффективный способ программирования на C++.

Никакие концепции не существует в абсолютной изоляции. Они сосуществуют и взаимодействуют друг с другом, и в этом проявляется вся их мощь. Попробуем выяснить, что такое автомобиль. Скоро у нас появятся понятия колес, двигателей, водителей, пешеходов, грузовиков, скорой помощи, дорог, масла, мотелей и т.д. Поскольку мы хотим использовать классы для представления концепций, то мы неизбежно сталкиваемся с вопросом о том, как можно программным способом отразить связи между концепциями? Ясно, что мы не можем непосредственно в языке отразить какие угодно абстрактные связи. Если бы даже могли, то вряд ли бы захотели это делать. Наши классы должны быть уже реальных концепций и должны быть более точно определены. Понятие производного класса и реализующий его

языковый механизм наследования классов призваны выразить понятие общности типов. Например, понятия круга и треугольника имеют общее в том отношении, что оба они суть фигуры; это значит, что понятие фигуры является общим для них понятием. Таким образом, нам нужно явным образом указать, что то общее, что есть у классов ***Circle*** (круг) и ***Triangle*** (треугольник), сосредоточено в классе ***Shape*** (фигура). Если же мы будем манипулировать лишь кругами и треугольниками без привлечения более общего понятия формы, то мы упустим что-то весьма существенное. В данной главе подробно изучаются смысл и значение этой простой идеи, лежащей в основе так называемого объектно-ориентированного программирования.

Соответствующие средства языка и приемы программирования изучаются постепенно в направлении от простого и конкретного к более изощренному и абстрактному. Для многих программистов это будет движение от хорошо знакомого к существенно менее известному. Это нельзя назвать движением от «старых плохих приемов программирования» к «единственно правильному стилю». Когда я указываю на ограничения одного метода в качестве мотивации для перехода к другому, я всегда это делаю в контексте некоторой задачи; для другой задачи и в ином контексте первый метод может снова стать наилучшим. Множество полезных программ было написано с применением всех рассматриваемых здесь приемов и методов. Главная цель состоит в том, чтобы помочь вам понять самую суть этих методов, дабы вы могли осознанно и со знанием дела осуществлять их выбор в контексте реальных задач.

Сначала я рассматриваю основные средства языка, поддерживающие объектно-ориентированное программирование. Затем эти средства будут использованы для хорошо структурированной разработки довольно большого практического примера. Иные средства поддержки объектно-ориентированного программирования, такие как множественное наследование и динамическая идентификация типов на этапе выполнения (run-time type identification), рассматриваются в главе 15.

12.2. Производные классы

Рассмотрим программу для обработки информации о сотрудниках (employees) некоторой фирмы. Такая программа может содержать следующую структуру данных по сотрудникам фирмы:

```
struct Employee
{
  string first_name, family_name;
  char middle_initial;
  Date hiring_date;
  short department;
  // ...
};
```

Теперь попытаемся определить структуру данных для менеджеров компании:

```
struct Manager
{
  Employee emp;          // общие сведения о менедже (как о сотруднике вообще)
```

```
  list<Employee*> group ; // подчиненные
  short level;
  // ...
};
```

Менеджеры также являются сотрудниками; соответствующие данные хранятся в поле ***emp*** класса ***Manager***. Для читателя программы это может быть вполне ясно, но компилятору и иным программным средствам ничто не говорит о том, что ***Manager*** является в то же самое время и ***Employee***. Указатель ***Manager**** имеет тип, отличный от указателя ***Employee****, так что нельзя просто так использовать один из них там, где требуется другой. В частности, нельзя поместить объект типа ***Manager*** в список объектов типа ***Employee***, не написав для этого специальный код. Можно, например, применить явное преобразование типа к ***Manager****, либо поместить в указанный список адрес члена ***emp***. Оба этих решения неэлегантны и могут запутать суть дела. Правильный подход — явно высказать утверждение, что ***Manager*** есть в то же время и ***Employee***:

```
struct Manager : public Employee
{
  set<Employee*> group;
  short level;
  // ...
};
```

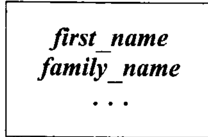
Здесь класс ***Manager*** указан как *производный* (*derived*) от класса ***Employee***, который, в свою очередь, является, тем самым, *базовым классом* (*base class*) для класса ***Manager***. В результате этого определения класс ***Manager*** имеет все члены класс ***Employee*** (***first_name***, ***department*** и т.д.) в дополнение к своим собственным членам (***group***, ***level*** и т.д.).

Рассмотренный только что механизм *наследования классов* (*classes inheritance*) графически отображается с помощью *стрелки, направленной от производного класса к базовому*, и показывающей, что именно *производный класс ссылается на базовый* (а не наоборот):

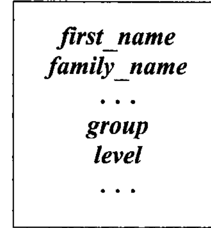
Часто говорят, что производный класс наследует свойства базового, и именно поэтому такая связь классов называется наследованием. Иногда базовый класс называют *суперклассом* (*superclass*), а производный класс — *подклассом* (*subclass*). Эта терминология часто смущает людей, которые понимают, что данные в объектах производного класса являются надмножеством данных объектов базового класса. Производный класс шире своего базового класса в том смысле, что он содержит больше данных и функций.

Популярная трактовка механизма наследования представляет объект производного класса как объект базового класса с дополнительной информацией, специфичной лишь для производного класса, добавленной в конец. Например:

Employee:



Manager:



Отсюда вытекает, что объекты типа ***Manager*** представляют собой подтип ***Employee***, ибо их можно использовать всюду, где допустимы ***Employee***. Например, теперь мы можем составить список сотрудников, некоторые из которых являются менеджерами:

```
void f(Manager m1, Employee e1)
{
 list<Employee*> elist;

 elist.push_front(&m1);
 elist.push_front(&e1);
 //...
}
```

Так как объект типа ***Manager*** является в то же время и ***Employee***, то можно использовать ***Manager**** и как ***Employee****. Обратное же необязательно верно — не каждый сотрудник является менеджером, так что ***Employee**** нельзя использовать вместо ***Manager****. В общем, если класс ***Derived*** является открытым наследником базового класса (§15.3) ***Base***, то переменную типа ***Derived**** можно присвоить переменной типа ***Base**** без необходимости в явном приведении типа. Обратное же присваивание требует явного преобразования типов. Например:

```
void g(Manager mm, Employee ee)
{
 Employee* pe = &mm;               // ok: каждый Manager есть Employee
 Manager* pm = &ee;                // error: не каждый Employee есть Manager

 pm->level = 2;                    // катастрофа: у объекта ee нету поля level

 pm=static_cast<Manager*>(pe);     // грубая сила работает, т.к. pe указует mm (Manager)
 pm->level = 2;                    // ok: pm указывает на mm, у которого есть level
}
```

Другими словами, с объектом производного класса можно обращаться как с объектом базового класса, *если манипулировать им через указатели или ссылки*. Обратное же неверно. Применение операций приведения ***static_cast*** и ***dynamic_cast*** обсуждается в §15.4.2.

Использование класса в качестве базового эквивалентно объявлению (неименованного) объекта этого класса. Следовательно, чтобы класс можно было использовать в качестве базового, он должен быть определен (§5.7):

```
class Eployee;                     // лишь объявление (не определение)
class Manager : public Employee    // error: Employee не определен
{
  // ...
};
```

12.2.1. Функции-члены

Столь простые структуры данных, как представленные выше ***Employee*** и ***Manager***, не слишком интересны и не слишком полезны. Нам нужно представить информацию с помощью надлежащего типа, предоставляющего все необходимые операции, и при этом не связываться с деталями конкретных представлений. Например:

```
class Employee
{
  string first_name, family_name;
  char middle_initial;
  // ...
public:
  void print() const;
  string full_name() const
  {return first_name+' '+middle_initial+' '+family_name;}
  // ...
};

class Manager : public Employee
{
  // ...
public:
  void print() const;
  // ...
};
```

Любой член производного класса может обращаться к любому открытому (а также защищенному — см. §15.3) члену базового класса, как будто бы последние были непосредственно объявлены в производном классе. Например:

```
void Manager::print() const
{
  cout<< "name is" << full_name() << '\n';
  // ...
}
```

Закрытые же члены базового класса напрямую *не доступны в производном классе*:

```
void Manager::print() const
{
  cout<< "name is" << family_name << '\n';    // error!
  // ...
}
```

Эта вторая версия ***Manager::print()*** компилироваться не будет. Члены производного класса не имеют специального разрешения на доступ к закрытым членам

базового класса, поэтому к ***family_name*** нельзя обращаться в теле функции ***Manager::print()***.

Для некоторых это является сюрпризом, но рассмотрим альтернативу: функция-член производного класса может иметь прямой доступ к закрытым членам базового класса. Концепция закрытых членов при этом стала бы *бессмысленной*, поскольку для доступа к закрытой части класса программист должен был бы всего лишь написать производный от него класс. Кроме того, чтобы отыскать все случаи использования закрытой части класса, недостаточно будет в таком случае просмотреть лишь функции-члены класса и его дружественные функции. Теперь нужно будет просматривать каждый исходный файл проекта на предмет обнаружения производных от исходного классов, тщательного изучения всех их функций-членов, а затем нужно будет выявить классы, производные от производных и так далее, и тому подобное. Это в самом лучшем случае утомительно, а часто и просто нереально. В базовых классах, где это возможно, нужно использовать ключевое слово ***protected*** вместо ***private***. Защищенные члены (protected) в отношении функций производных классов ведут себя как открытые, а в остальных случаях — как закрытые (§15.3).

Как правило, самым простым решением является использование в производном классе лишь открытых членов базового класса. Например:

```
void Manager::print() const
{
  Employee::print();     // вывести общую информацию
                         // (характерную для любого сотрудника)
  cout<< level;          // вывести информацию, специфичную для типа Manager
}
```

Обратите внимание на то, что нужно использовать операцию ::, поскольку функция ***print()*** переопределяется (redefined) в классе ***Manager***. Очень опрометчиво написать следующее:

```
void Manager::print() const
{
  // печатаем специфичную для типа Manager информацию:
  print();        // oops!

}
```

ибо при этом порождается неожиданная рекурсия.

12.2.2. Конструкторы и деструкторы

Некоторые производные классы нуждаются в конструкторах. Если в базовом классе определены конструкторы, то их тоже нужно вызывать. Конструкторы по умолчанию могут вызываться неявно. Остальные же типы конструкторов базового класса должны вызываться явно. Например:

```
class Employee
{
  string first_name, family_name;
  short department;
  //...
```

```
public:
  Employee(const string& n, int d);
  // ...
};

class Manager : public Employee
{
  list<Employee*> group;    // подчиненные
  short level;
  // ...

public:
  Manager(const string& n, int d, int lvl);
  // ...
};
```

Аргументы для конструктора базового класса указываются в аргументах конструктора производного класса. В этом отношении, базовый класс функционирует в точности как член производного класса (§10.4.6). Например:

```
Employee::Employee(const string& n, int d)
      : family_name(n), department(d)    // инициализация членов класса Employee
{
  // ...
}

Manager::Manager(const string& n, int d, int lvl)
      : Employee(n,d),                   // инициализация членов базового класса
        level(lvl)                       // инициализация членов класса Manager
{
  // ...
}
```

Конструктор производного класса может задавать инициализаторы только для своих членов и непосредственных базовых классов (он не может явным образом инициализировать члены базового класса). Например:

```
Manager::Manager(const string& n, int d, int lvl)
      : family_name(n),                  // error: family_name не объявлен в Manager
        department(d),                   // error: department не объявлен в Manager
        level(lvl)
{
  // ...
}
```

Здесь содержатся три ошибки: не удается вызвать конструктор базового класса ***Employee***, а также дважды осуществляется ошибочная попытка прямой инициализации членов базового класса.

Объекты классов конструируются снизу-вверх: сначала базовый класс, затем инициализируются члены, и только потом остальная инициализация производного класса. Уничтожение объектов выполняется в обратном порядке. Конструирование членов (указанных в списке инициализации конструктора) выполняется строго в порядке появления их объявлений в определении класса, уничтожение же производится точно в обратном порядке. См. также §10.4.6, §15.2.4.1 и §15.4.3.

12.2.3. Копирование

Копирование классовых объектов определяется копирующим конструктором и операцией присваивания (§10.4.4.1). Рассмотрим пример:

```
class Employee
{
  // ...
  Employee& operator= (const Employee&) ;
  Employee (const Employee&) ;
};

void f(const Manager& m)
{
  Employee e = m;          // конструируем e из Employee-части m
  e = m;                   // присваиваем Employee-часть m объекту e
}
```

Поскольку копирующие функции класса ***Employee*** ничего не знают о классе Manager, то копируется лишь часть объекта типа ***Manager*** — та, что достается ему от базового класса ***Employee***. Этот эффект часто называют *срезкой* (*slicing*), и он может оказаться причиной недоразумений и ошибок. Одной из причин передачи указателей на объекты классовых иерархий наследования является желание избежать срезки. Другими причинами служат желание обеспечить полиморфное поведение (§2.5.4, §12.2.6) и достичь эффективности.

Помните, что если вы не программируете явно операцию присваивания, то компилятор предоставляет собственный вариант этой операции (§11.7). Это подразумевает, что *операции присваивания не наследуются. Конструкторы тоже никогда не наследуются.*

12.2.4. Иерархии классов

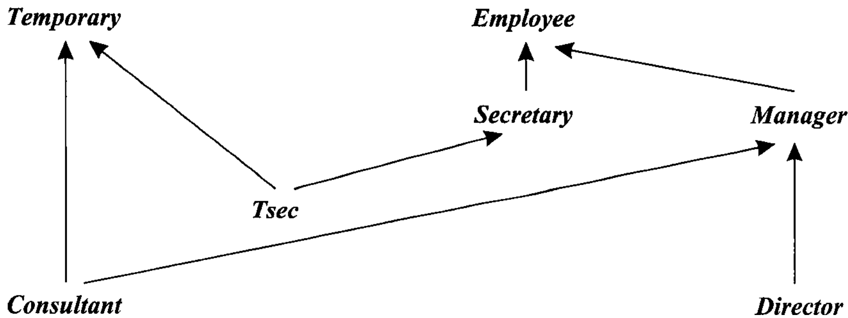
Производный класс, в свою очередь, тоже может быть базовым классом. Например:

```
class Employee { /*...*/ };
class Manager : public Employee{ /*...*/ };
class Director : public Manager{ /*...*/ };
```

Такой набор связанных классов традиционно называют классовой иерархией наследования или просто *классовой иерархией* (*class hierarchy*). Ее чаще всего изображают в виде дерева, но она может иметь и более общую структуру графа. Например, для классов

```
class Temporary{ /*...*/ };
class Secretary : public Employee{ /*...*/ };
class Tsec : public Temporary, public Secretary{ /*...*/ };
class Consultant : public Temporary, public Manager{ /*...*/ };
```

в графической форме иерархия имеет вид:



Таким образом, как детально объясняется в §15.2, на C++ можно сформировать направленный ациклический граф классов.

12.2.5. Поля типа

Чтобы использовать производные классы в качестве чего-то большего, чем просто удобный способ сокращения исходного кода, мы должны решить следующую проблему: как определить истинный (производный) тип объекта, адресуемого указателем типа ***Base**** (то есть указателем на базовый класс)? Существуют четыре фундаментальных способа решения этой проблемы:

1. Гарантировать, что адресуются лишь объекты единственного типа (§2.7, глава 13).
2. Поместить специальное поле типа в базовый класс, чтобы функции могли его проверять.
3. Использовать ***dynamic_cast*** (§15.4.2, §15.4.5).
4. Использовать виртуальные функции (§2.5.5, §12.2.6).

Указатели на базовые классы широко используются при проектировании *контейнерных классов* (*container classes*), таких как множества (set), вектора (vector) и списки (list). В таких случаях решение [1] приведет к однородному списку, то есть списку объектов одного и того же типа. Решения [2], [3] и [4] позволяют строить списки разнородных объектов, то есть списки указателей, настроенных на объекты разных типов. Решение [3] является вариантом решения [2], которое поддерживается языком C++ непосредственно. А решение [4] является специальным вариантом решения [2], безопасным по отношению к типам. Комбинация решений [1] и [4] особо интересна и чрезвычайно мощна, и почти всегда способствует более ясному коду, чем код, порождаемый решениями [2] и [3].

Рассмотрим сначала решение, основанное на полях типа с тем, чтобы понять, почему его в большинстве случаев лучше избегать. Наш пример с сотрудниками и менеджерами можно переопределить следующим образом:

```
struct Employee
{
  enum Empl_type {M, E};
  Empl_type type;

  Employee() : type(E) { }
```

```
    string first_name, family_name;
    char middle_initial;

    Date hiring_date;
    short department;
    // ...
};

struct Manager : public Employee
{
    Manager () {type = M; }

    list<Employee*> group;     // подчиненные
    short level;
    // ...
};
```

Теперь мы можем написать функцию, выводящую информацию о каждом сотруднике:

```
void print_employee (const Employee* e)
{
    switch (e->type)
    {
        case Employee::E:
            cout<< e->family_name<< '\t'<< e->department<< '\n';
            // ...
            break;
        case Employee::M:
        {
            cout<< e->family_name<< '\t'<< e->department<< '\n';
            // ...
            const Manager* p = static_cast<const Manager*> (e);
            cout<< "level" <<p->level<< '\n';
            // ...
            break;
        }
    }
}
```

Используем эту функцию для вывода элементов списка:

```
void print_list (const list<Employee*>& elist)
{
    for (list<Employee*>::const_iterator p = elist.begin (); p!=elist.end (); ++p)
        print_employee (*p);
}
```

Все это прекрасно работает, особенно в маленьких программах, поддерживаемых единственным программистом. В то же время, представленное решение имеет и фундаментальную слабость, так как зависит от манипуляций с типами, которые не контролируются компилятором. Положение становится еще хуже, если функции, вроде ***print_employee*** () пытаются извлечь пользу из общности типов:

```
void print_employee (const Employee* e)
{
  cout<< e->family_name<< '\t'<< e->department<< '\n';
  //...
  if( e->type == Employee::M)
  {
    const Manager* p = static_cast<const Manager*> (e);
    cout<< "level" << p->level<< '\n';
    //...
  }
}
```

Поиск всех проверок типа, глубоко запрятанных в коде больших функций, да еще и при наличии большого числа производных классов, весьма утомителен. Даже если все проверки найдены, бывает непросто понять, что на самом деле происходит. Более того, если добавляется новый тип сотрудников (производный от **Employee** класс), то приходится вносить изменения во все ключевые функции системы — те, что проверяют поля типа. После внесения изменений программист должен проверить все функции, потенциально зависящие от проверок типа. Это подразумевает, что программист должен иметь доступ к ключевым частям исходного кода программы, а также дополнительные усилия по тестированию программы. В итоге становится понятным, что наличие проверок типа служит отчетливым сигналом о необходимости улучшения кода.

Другими словами, техника применения полей типа способствует ошибкам и усложняет сопровождение. Проблема усиливается чрезвычайно по мере роста размеров программ, ибо поля типа нарушают все принципы модульности и сокрытия данных. Каждая функция, использующая поля типа, должна знать детали представления всех производных классов иерархии (с базовым классом, содержащим поле типа).

Также складывается впечатление, что наличие общих данных, доступных из всех классов иерархии наследования, провоцирует программистов на увеличение числа таких данных. В итоге, общий базовый класс становится чем-то вроде склада всякой «полезной информации». Это, в свою очередь, порождает крайне нежелательные связи между реализациями базовых и производных классов. Ясный дизайн и простота сопровождения требуют, чтобы разные сущности представлялись раздельно, а взаимозависимости между ними минимизировались.

12.2.6. Виртуальные функции

Виртуальные функции решают проблему полей типа, предоставляя программисту возможность определить в базом классе функции, подлежащие замещению в каждом производном классе. Компилятор и загрузчик гарантируют правильное соответствие между объектами и применяемыми к ним функциями. Например:

```
class Employee
{
  string first_name, family_name;
  short department;
  //...
public:
```

```
    Employee(const string& name, int dept);
    virtual void print() const;
    //...
};
```

Ключевое слов ***virtual*** указывает, что функция ***print***() символизирует общий интерфейс к набору одноименных функций, определенных в базовом и производных от него классах. Если такие функции действительно определены в производных классах, компилятор гарантирует вызов своего варианта функции ***print***() для каждого объекта классовой иерархии.

Чтобы объявление виртуальной функции действительно работало в качестве интерфейса к семейству функций, определенных в базовом и производных классах, *типы аргументов* всех функций *должны быть одинаковыми*, а для возвращаемого значения допускаются лишь незначительные отличия (§15.6.2). Виртуальные функции-члены иногда называют методами[1].

Виртуальная функция *обязана быть определена* в классе, в котором она впервые объявляется (за исключением случаев чисто виртуальных функций; см. §12.3). Например:

```
void Employee::print() const
{
    cout<<family_name<<'\t'<<department<<'\n';
    //...
}
```

Разумеется, виртуальную функцию можно использовать и в случаях, когда у класса отсутствуют производные классы. Производный класс, который не нуждается в собственном варианте функции, *не обязан замещать виртуальную функцию*, определенную в базовом классе. Определяя производный класс, определите и новый вариант виртуальной функции, если он, конечно, нужен. Например:

```
class Manager :public Employee
{
    list<Employee*> group;
    short level;
    //...
public:
    Manager(const string& name, int dept, int lvl);
    void print() const;
    //...
};

void Manager::print() const
{
    Employee::print();

    cout<<"\tlevel"<<level<<'\n';
    //...
}
```

[1] Чаще всего методами называют любые классовые функции-члены. — *Прим. ред.*

Функция производного класса, имеющая то же имя и те же параметры, что и виртуальная функция, определенная в базовом классе, *замещает* (*override*) вариант от базового класса. За исключением случаев прямых указаний на конкретный вариант используемой виртуальной функции (например, ***Employee::print()***), выбирается замещаемая для класса объекта функция.

Теперь отпадает нужда в глобальной функции ***print_employee()*** (§12.2.5), поскольку ее место занято функциями-членами ***print()***. Список сотрудников теперь выводится следующим образом:

```
void print_list (const list<Employee*>& s)
{
  for (list<Employee*>::const_iterator p = s.begin();p!=s.end();++p)    //см. §2.7.2
    (*p)->print();
}
```

Или еще проще:

```
void print_list (const list<Employee*>& s)
{
  for_each (s.begin(),s.end(),mem_fun(&Employee::print));             // см. §3.8.5
}
```

Информация по каждому сотруднику выводится точно в соответствии с его настоящим статусом. Вот иллюстрация к сказанному, где функция ***main()***:

```
int main ()
{
  Employee e ("Brown",1234);
  Manager m ("Smith",1234,2);
  list<Employee*> empl;
  empl.push_front(&e);                                                // см. §2.5.4
  empl.push_front (&m);
  print_list(empl);
}
```

порождает следующий вывод:

```
Smith 1234
    level 2
Brown 1234
```

Заметим, что все это работает даже в случае, когда функция ***print_list()*** написана и откомпилирована до того, как класс ***Manager*** был разработан (и даже задуман). Вот это действительно ключевой аспект работы с классами. При правильном использовании он служит краеугольным камнем объектно-ориентированного проектирования и придает стабильности эволюционирующим проектам.

«Правильное» поведения виртуальных функций ***print()*** для любых сотрудников компании (то есть для любых объектов типа ***Employee***, ***Manager*** и прочих производных классов) называется *полиморфизмом* (*polymorphism*). Тип с виртуальными функциями называется *полиморфным типом* (*polymorphic type*). Для практической реализации полиморфного поведения в языке C++ тип объектов должен быть полиморфным, а сами объекты должны адресоваться указателями или ссылками. Когда же с объектами работают напрямую (а не косвенно через указатели или

ссылки), то их тип известен компилятору заранее и никакого полиморфизма просто не требуется.

Ясно, что для реализации полиморфизма, компилятор должен хранить с каждым объектом полиморфной классовой иерархии дополнительную информацию о типе, чтобы иметь возможность вызвать правильную версию виртуальной функции. В типичной реализации достаточно дополнительной памяти для хранения указателя (§2.5.5). Эта дополнительная память относится только к объектам классов с виртуальными функциями, а вовсе не ко всем объектам любых классовых иерархий. Отметим, что в рассмотренном выше случае, когда применяется поле типа, величина требуемой избыточной памяти ничуть не меньше.

Вызов функции с применением операции разрешения области видимости `::`, как это было сделано в ***Manager*::*print*()**, гарантирует «выключение» механизма виртуальных функций. В противном случае, вызов ***Manager*::*print*()** привел бы к бесконечной рекурсии. Использование квалифицированных имен имеет еще один положительный эффект. Если виртуальная функция объявлена встраиваемой (что нередко встречается), то применение операции `::` позволяет действительно встраивать ее вызов. А это помогает программисту уладить деликатную проблему вызова виртуальной функции из другой виртуальной функции для того же самого объекта. Функция ***Manager*::*print*()** как раз служит наглядным примером. Поскольку тип объекта фиксирован уже при вызове ***Manager*::*print*()**, нет нужды определять его снова для вызова ***Employee*::*print*()**.

Стоит не забывать, что очевидной и традиционной реализацией механизма вызова виртуальных функций является косвенный вызов (§2.5.5), высокая эффективность которого не должна удерживать программиста от применения виртуальных функций там, где приемлем обычный функциональный вызов.

12.3. Абстрактные классы

Многие классы схожи с классом ***Employee*** в том отношении, что они полезны и сами по себе, и как базовые для производных классов. Для подобных классов описанные в предыдущем разделе методики вполне достаточны. Но не все классы вписываются в такую схему работы. Некоторые классы, вроде класса ***Shape***, представляют абстрактные концепции, для которых реальных объектов не существует. Класс ***Shape*** имеет смысл и полезен исключительно как абстрактный базовый класс для порождения конкретных производных классов. Это наглядно видно из того факта, что невозможно разумным образом определить его виртуальные функции:

```
class Shape
{
public:
  virtual void rotate(int) {error("Shape::rotate");}   // не элегантно
  virtual void draw() {error("Shape::draw");}
  // ...
};
```

Создание таких «бесформенных» объектов хотя и допустимо, но лишено всякого смысла:

```
Shape s;    // глупо: "shapeless shape"
```

Действительно, какой смысл в объекте, любая операция над которым порождает лишь сообщение об ошибке.

Лучше объявить виртуальные функции класса ***Shape*** как *чисто виртуальные* (*pure virtual*) с помощью инициализатора ***=0***:

```
class Shape                          // абстрактный класс
{
public:
  virtual void rotate(int) = 0;      // чисто виртуальная функция
  virtual void draw() = 0;           // чисто виртуальная функция
  virtual bool is_closed() = 0;      // чисто виртуальная функция
  // ...
};
```

Класс с одной или несколькими чисто виртуальными функциями является *абстрактным классом* (*abstract class*), объекты которого создавать недопустимо:

```
Shape s;                             // error: переменная абстрактного класса Shape
```

Абстрактный класс может использоваться только как интерфейсный и базовый для других классов. Например:

```
class Point{ /* ... */ };

class Circle : public Shape
{
public:
  void rotate(int) { }               // замещает Shape::rotate
  void draw();                       // замещает Shape::draw
  bool is_closed() {return true; }   // замещает Shape::is_closed

  Circle(Point p, int r);

private:
  Point center;
  int radius;
};
```

Чисто виртуальная функция, не определенная и в производном классе, остается чисто виртуальной, и поэтому такой производный класс также является абстрактным. Это открывает нам возможности постепенного осуществления реализации классовых иерархий:

```
class Polygon : public Shape         // абстрактный класс
{
public:
  bool is_closed() {return true; }   // замещает Shape::is_closed
  // ... draw and rotate не замещаются
};

Polygon b;                           // error: объект абстрактного класса Polygon

class Irregular_polygon : public Polygon
{
  list<Point> lp;
```

```
public:
  void draw() ;                    // замещает Shape::draw
  void rotate(int) ;               // замещает Shape::rotate
  // ...
};

Irregular_polygon poly(some_points) ; // ok: (если есть подходящий конструктор)
```

Важным примером применения абстрактных классов является предоставление интерфейса без какой-либо реализации. Например, операционная система может скрывать детали реализации аппаратных драйверов за вот таким абстрактным интерфейсом:

```
class Character_device
{
public:
  virtual int open(int opt)=0;
  virtual int close(int opt)=0;
  virtual int read(char* p, int n)=0;
  virtual int write(const char* p, int n)=0;
  virtual int ioctl(int...)=0;
  virtual ~Character_device() { }     // виртуальный деструктор
};
```

Мы можем определить драйверы в форме классов, производных от ***Character_device***, после чего управлять ими через указанный стандартный интерфейс. Важность виртуальных деструкторов разъясняется в §12.4.2.

Теперь, после представления абстрактных классов, мы знакомы со всем инструментарием, позволяющим писать модульные программы, использующие классы в качестве строительных блоков.

12.4. Проектирование иерархий классов

Рассмотрим несложную проектную задачу: ввод в программу целого значения через пользовательский интерфейс. Это можно сделать огромным количеством способов. Чтобы позиционировать нашу задачу в рамках этого множества, и чтобы проанализировать различные возможные проектные решения, начнем с определения программной модели ввода. Оставим на потом детали, необходимые для ее конкретной реализации в рамках реальных пользовательских интерфейсов.

Идея состоит в создании класса ***Ival_box***, который знает допустимый диапазон вводимых значений. Программа может запросить ***Ival_box*** об этом значении, а также предупредить пользователя. Также программа может попросить ***Ival_box*** сообщить, не вводил ли пользователь новых значений после того, как программа получила предыдущее значение ввода.

Поскольку можно по-разному реализовывать эту общую идею, мы предполагаем, что будет множество конкретных разновидностей элементов ввода типа ***Ival_box***, таких как ползунки, диалоговые окна, элементы для голосового ввода и т.д.

В общем, мы хотим реализовать «виртуальную систему пользовательского ввода» для использования в разных приложениях. Она будет демонстрировать часть

функциональности, реализованной в настоящих системах пользовательского интерфейса. Ее желательно реализовать в широком наборе операционных систем, так что нельзя забывать о переносимости кода. Естественно, наш подход не единственно возможный. Я выбрал его потому, что он достаточно общий и позволяет продемонстрировать широкий набор методов и технологий проектирования. Эти методы не только использовались для построения реальных систем пользовательского интерфейса, но они вообще выходят далеко за рамки интерфейсных систем.

12.4.1. Традиционные иерархии классов

Наше первое решение сводится к традиционной иерархии классов, типичной для языков Simula, Smalltalk и старых версий C++.

Класс *Ival_box* определяет базисный интерфейс для всех элементов пользовательского ввода и задает его реализацию, которую более специфические элементы могут переопределять. Кроме того, мы объявляем здесь данные, необходимые для формирования любого элемента ввода:

```
class Ival_box
{
protected:
  int val;
  int low, high;
  bool changed;

public:
  Ival_box (int ll, int hh) {changed = false; val = low = ll; high = hh; }

  virtual int get_value () {changed = false; return val; }       // для приложения
  virtul void set_value (int i) {changed=true; val=i; }          // для пользователей
  virtual void reset_value (int i) {changed=false; val=i; }      // для приложения

  virtual void prompt () { }
  virtual bool was_changed () const {return changed; }
};
```

Представленная здесь реализация функций довольно небрежна и нацелена лишь на демонстрацию основных намерений. В реальном коде осуществлялась бы хоть какая-то проверка введенных значений.

Данный класс можно использовать следующим образом:

```
void interact (Ival_box* pb)
{
  pb->prompt () ;                          // оповестить пользователя
  // ...
  if (pb->was_changed () )
  {
    int i=pb->get_value () ;
    // новое значение; что-нибудь делаем
  }
  else
  {
    // старое значение подходит; делаем что-нибудь еще
  }
```

```
  // ...
}

void some_fct ()
{
  Ival_box* p1 = new Ival_slider (0, 5) ;     // Ival_slider наследует от Ival_box
  interact (p1) ;

  Ival_box* p2 = new Ival_dial (1, 12) ;
  interact (p2) ;
}
```

Большая часть кода написана в стиле, использующем доступ к элементу ввода через указатель типа ***Ival_box**** (см. функцию ***interact***()). Поэтому приложению нет нужды знать обо всех потенциально возможных конкретных элементах ввода. Знание же об этих специфических элементах требуется лишь в строго ограниченных частях кода (небольшом количестве функций), имеющих дело с созданием объектов соответствующих типов. Это помогает в значительной степени изолировать пользователя от изменений в реализациях производных классов. Большая часть кода может не обращать внимания на существование множества различных интерфейсных элементов ввода.

Чтобы упростить изложение основных идей проектирования, я сознательно не рассматриваю вопрос о том, как именно программа ожидает ввод. Возможно программа действительно ожидает ввода в функции ***get_value***(), а может она ассоциирует ***Ival_box*** с некоторым событием и готовится реагировать на него функцией обратного вызова (callback function); также возможно, что она запускает код ***Ival_box*** в отдельном потоке, а потом опрашивает его состояние. Конечно, такие вопросы предельно важны в разработках конкретных систем пользовательского интерфейса. Однако даже минимальное обсуждение этих вопросов отвлекло бы нас от изучения средств языка и общих методов программирования, которые совсем не ограничиваются одними лишь пользовательскими интерфейсами. Они применимы к очень широкому кругу задач.

Конкретные элементы ввода определяются как производные от ***Ival_box*** классы. Например:

```
class Ival_slider : public Ival_box
{
  // графические атрибуты, определяющие вид ползунка (slider) и т.д.
public:
  Ival_slider (int, int) ;
  int get_value () ;
  void prompt () ;
} ;
```

Члены данных класса ***Ival_box*** были объявлены защищенными с целью предоставления прямого к ним доступа из производных классов. Таким образом, ***Ival_slider::get_value***() может держать значение в ***Ival_box::val***. Напоминаем, что защищенные члены доступны самому классу и его производным классам, но не обычному клиентскому коду (§15.3).

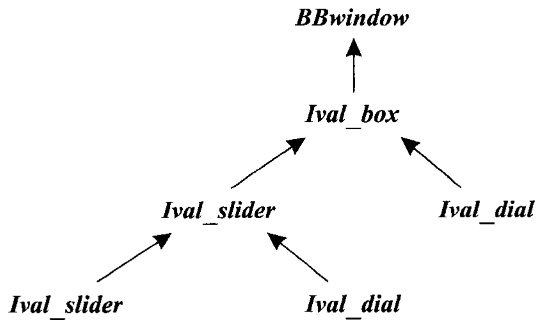
В дополнение к ***Ival_slider*** мы определим и другие конкретные элементы ввода, реализующие специфические варианты общей концепции ***Ival_box***. Это может быть

Ival_dial, позволяющий выбрать значение с помощью вращающейся «ручки»; ***Flashing_ival_slider***, который мерцает при вызове ***prompt***(); ***Popup_ival_slider***, который реагирует на вызов ***prompt***() «всплытием» на экране где-нибудь в заметном месте, чтобы пользователь не мог его проигнорировать.

Но откуда при этом возьмется графическая начинка для элементов ввода? Большинство систем графического интерфейса пользователя предоставляют класс, определяющий основные свойства отображения элементов на экране. Тогда, если мы, например, воспользуемся наработками фирмы «Big Bucks Inc.», то все наши классы должны наследовать от стороннего класса ***BBwindow***. Для этого достаточно сделать класс ***Ival_box*** производным от ***BBwindow***. Тогда, например, любой элемент ввода может быть «помещен» на экран и при этом сразу будет подчиняться определенным графическим правилам (может изменять свой визуальный размер, может быть отбуксирован с помощью мыши и т.д.), принятым в ***BBwindow***. Наша классовая иерархия примет при этом следующий вид:

```
Class Ival_box : public BBwindow{ /* ... */ };      // переписан для использования
class Ival_slider : public Ival_box{ /* ... */ };   // с BBwindo
class Ival_diall : public Ival_box{ /* ... */ };
class Flashing_ival_slider : public Ival_slider{ /* ... */ };
class Popup_ival_slider : public Ival_slider{ /* ... */ };
```

или в графической форме:



12.4.1.1. Критика

Представленный дизайн во многих случаях работает замечательно, а соответствующая классовая иерархия хорошо подходит для решения многих проблем. Тем не менее, отдельные детали все же заставляют нас рассмотреть и другие альтернативы.

Мы сделали ***BBwindow*** базовым классом для ***Ival_box***. Это не совсем верно. Использование ***BBwindow*** не входит в понимание существа ***Ival_box***; это всего лишь деталь реализации. Однако применение этого класса в качестве базового для ***Ival_box*** превратило его в важнейший элемент дизайна всей системы. Это в каких-то случаях может быть приемлемым, например, если мы накрепко связываем свои разработки с «Big Bucks Inc.». Но что, если мы захотим визуализировать наш ***Ival_box*** в рамках графической системы от «Imperial Bananas», «Liberated Software» или «Compiler Whizzes»? Нам тогда придется поддерживать четыре различные версии программы:

```
class Ival_box : public BBwindow{ /* ... */ };    // BB version
class Ival_box : public CWwindow{ /* ... */ };    // CW version
class Ival_box : public IBwindow{ /* ... */ };    // IB version
class Ival_box : public LSwindow{ /* ... */ };    // LS version
```

Множество версий программы может стать настоящим кошмаром.

Другая проблема состоит в том, что каждый производный класс разделяет данные, объявленные в *Ival_box*. Эти данные безусловно являются деталью реализации, вкравшейся в интерфейс *Ival_box*. С практической же точки зрения, во многих случаях эти данные неадекватны. Например, элемент *Ival_slider* не нуждается в специальном хранении связанного с ним значения — его можно легко вычислить по положению ползунка этого элемента во время вызова *get_value*(). С другой стороны, хранить связанные, но различные наборы данных — это значит нарываться на неприятности: рано или поздно они окажутся рассогласованными. Кроме того, многие новички склонны объявлять данные защищенными, что усложняет сопровождение. Данные лучше объявлять закрытыми, чтобы разработчики производных классов не пытались запутывать их в один клубок со своими данными. А еще лучше объявлять данные в производных классах, где их можно определить наиболее точным образом, и они не будут усложнять жизнь разработчикам иных (несвязанных) производных классов. Почти всегда интерфейс должен содержать только функции, типы и константы.

Преимущество наследования от *BBwindow* состоит в том, что его возможности сразу становятся доступными пользователям *Ival_box*. К сожалению, это же означает, что в случае изменений в *BBwindow* пользователю придется перекомпилироваться (или даже внести изменения в исходный код), чтобы справиться с последствиями этих изменений. В частности, для большинства реализаций C++ изменения в размере базового класса автоматически влекут за собой необходимость перекомпиляции всех производных классов.

И наконец, наша программа может работать в смешанных средах, где сосуществуют различные системы пользовательского интерфейса. Причина может заключаться в том, что две системы каким-то образом совместно используют экран, или что нашей программе потребовалось взаимодействовать с пользователями других систем. Жесткое встраивание какого-либо пользовательского интерфейса в основу классовой иерархии для нашего единственного интерфейса *Ival_box* не является гибким решением в таких ситуациях.

12.4.2. Абстрактные классы

Итак, начнем сначала и построим новую иерархию классов, которая решает проблемы, вытекающие из представленной выше критики традиционной иерархии:

1. Система пользовательского интерфейса должна быть деталью реализации, скрытой от пользователей, не желающих знать о ней.
2. Класс *Ival_box* не должен содержать данных.
3. Изменения в системе пользовательского интерфейса не должны требовать перекомпиляции кода, использующего классы иерархии *Ival_box*.
4. Различные варианты *Ival_box* для разных систем пользовательского интерфейса должны иметь возможность сосуществовать в нашей программе.

Для достижения поставленных целей существует несколько альтернативных подходов. Здесь я представляю один из них, наилучшим образом вписывающийся в возможности C++.

Сначала я определяю ***Ival_box*** как чистый интерфейс:

```
class Ival_box
{
public:
  virtual int get_value () = 0;
  virtual void set_value (int i) = 0;
  virtual void reset_value (int i) = 0;
  virtual void prompt () = 0;
  virtual bool was_changed () const = 0;
  virtual~Ival_box () { }
};
```

Это намного понятнее, чем прежнее объявление ***Ival_box***. Данные исчезли вместе с упрощенными реализациями функций-членов. Ушел также и конструктор, так как отсутствуют данные, подлежащие инициализации. Вместо этого я добавил виртуальный деструктор для того, чтобы гарантировать правильную очистку данных, определяемых в производных классах.

Определение ***Ival_slider*** может выглядеть следующим образом:

```
class Ival_slider : public Ival_box, protected BBwinow
{
public:
  Ival_slider (int, int) ;
  ~Ival_slider () ;

  int get_value () ;
  void set_value (int i) ;
  // ...
protected:
  // функции, замещающие виртуальные функции,
  // например, BBwindow::draw(), BBwindow::mouse1hit()
private:
  // данные для slider
};
```

Класс ***Ival_slider*** наследуется от абстрактного класса (***Ival_box***), требующего реализовать его чисто виртуальные функции. Кроме того, ***Ival_slider*** наследуется от ***BBwindow***, предоставляющего необходимые для реализации средства. Поскольку ***Ival_box*** объявляет интерфейс для производных классов, наследование от него выполняется в открытом режиме (с применением ключевого слова ***public***). А поскольку ***BBwindow*** есть просто средство реализации, то наследование от него выполняется в защищенном режиме (с использованием ключевого слова protected — см. §15.3.2). Из этого следует, что программист, использующий ***Ival_slider*** не может напрямую воспользоваться средствами ***BBwindow***. Интерфейс ***Ival_slider*** состоит из открыто унаследованного интерфейса ***Ival_box*** плюс то, что явно объявляет сам ***Ival_slider***. Я использовал защищенное наследование вместо более ограничительного (и обычно более безопасного) закрытого, чтобы сделать ***BBwindow*** доступным для классов, производных от ***Ival_slider***.

Непосредственное наследование от более, чем одного класса, называется *множественным наследованием* (*multiple inheritance*) (§15.2). Отметим, что ***Ival_slider*** должен замещать виртуальные функции из ***Ival_box*** и ***BBwindow***. Поэтому он прямо или косвенно должен наследовать от обоих этих классов. Как показано в §12.4.1.1, возможно косвенное наследование от ***BBwindow***, когда последний служит базовым классом для ***Ival_box***, но это имеет свои нежелательные побочные эффекты. Объявление же поля с типом ***BBwindow*** членом класса ***Ival_box*** не подходит потому, что класс не замещает виртуальные функции своих членов (§24.3.4). Представление графических средств (окна) в виде члена ***Ival_box*** с типом ***BBwindow**** приводит к совершенно другому стилю проектирования со своими собственными компромиссными «за» и «против» (§12.7[14], §25.7).

Интересно, что новое объявление ***Ival_slider*** позволяет написать код приложения абсолютно так же, как раньше. Все, что мы сделали, это реструктурировали детали реализации более логичным образом.

Многие классы требуют некоторой формы очистки данных перед уничтожением объекта. Поскольку абстрактный класс ***Ival_box*** не может знать, требуется ли очистка в производных классах, то ему лучше заранее предположить, что требуется. Мы гарантируем надлежащую очистку, объявляя виртуальный деструктор ***Ival_box::~Ival_box()*** в базовом классе, и замещая его в производных классах. Например:

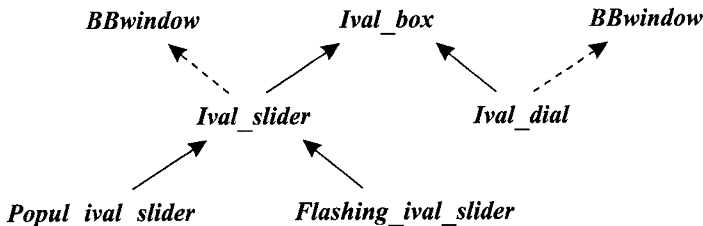
```
void f(Ival_box* p)
{
  // ...
  delet p;
}
```

Операция ***delete*** явным образом уничтожает объект, адресуемый указателем ***p***. Мы не можем знать, какому точно классу принадлежит объект, адресуемый указателем ***p***, но благодаря виртуальному деструктору из ***Ival_box*** надлежащий деструктор (опционально) будет вызван.

Теперь иерархию ***Ival_box*** можно определить следующим образом:

```
class Ival_box {/* ... */};
class Ival_slider : public Ival_box, protected BBwindow { /* ... */ };
class Ival_dial : public Ival_box, protected BBwindow { /* ... */ };
class Flashing_ival_slider : public Ival_slider { /* ... */ };
class Popup_ival_slider : public Ival_slider { /* ... */ };
```

или в графической форме:



Я использую пунктирную линию для отображения защищенного наследования. Для обычных пользователей это деталь реализации.

12.4.3. Альтернативные реализации

Полученный дизайн яснее, чем традиционный, и поддерживать его легче. К тому же, он не менее эффективный. Но пока что он по-прежнему не решает проблему со многими версиями:

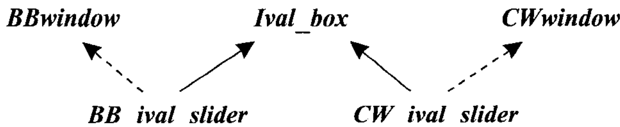
```
class Ival_box { /* ... */ };     // common
class Ival_slider : public Ival_box, protected BBwindow{/* ... */};     // для BB
class Ival_slider : public Ival_box, protected CWwindow{/* ... */};     // для CW
// ...
```

И он не позволяет ***Ival_slider*** для ***BBwindow*** сосуществовать с ***Ival_slider*** для ***CWwindow***, даже если сами эти системы пользовательского интерфейса сосуществуют.

Очевидным решением проблемы является определение нескольких вариантов ***Ival_slider*** с разными именами:

```
class Ival_box { /* ... */ };
class BB_ival_slider : public Ival_box, protected BBwindow { /* ... */ };
class CW_ival_slider : public Ival_box, protected CWwindow { /* ... */ };
// ...
```

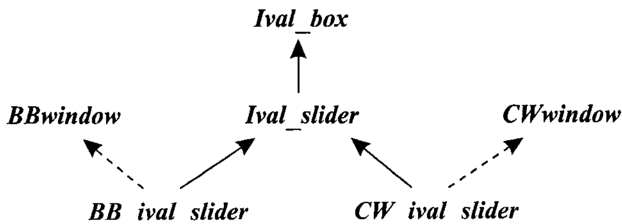
или графически:



Чтобы еще более изолировать наши классы от деталей реализации, мы можем ввести абстрактный класс ***Ival_slider***, который наследует от ***Ival_box***, а системно-зависимые элементы наследовать следующим образом:

```
class Ival_box { /* ... */ };
class Ival_slider : public Ival_box { /* ... */ };
class BB_ival_slider : public Ival_slider, protected BBwindow { /* ... */ };
class CW_ival_slider : public Ival_slider, protected CWwindow { /* ... */ };
```

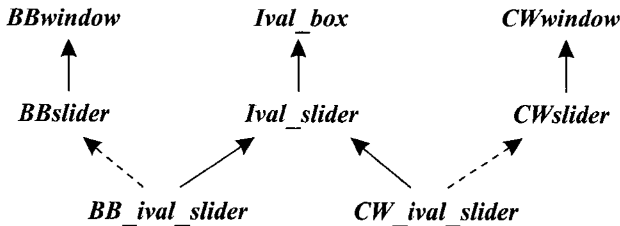
или графически:



Как правило, ситуация только улучшается, если системно-зависимые классы также выстроены в иерархию. Например, если у фирму «Big Bucks Inc.» имеется класс ползунков (slider class), то мы производим собственный элемент, наследуя от этого ползунка:

```
class BB_ival_slider : public Ival_slider, protected BBslider { /* ... */ };
class CW_ival_slider : public Ival_slider, protected CWslider { /* ... */ };
// ...
```

или в графической форме:



Последнее улучшение особо велико там, где наши абстракции классов не слишком отличаются от абстракций программной системы, обеспечивающей графическую реализацию. Тогда объем программирования уменьшается и сводится к необходимости установления соответствия между аналогичными концепциями. Наследование же от общих базовых классов, таких как ***BBwindow***, производится редко.

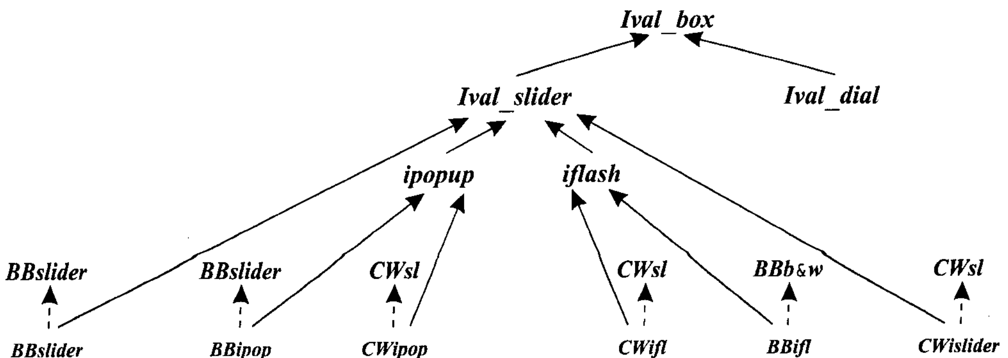
Окончательная иерархия будет состоять из нашей оригинальной иерархии интерфейсов, ориентированных на наше приложение:

```
class Ival_box { /* ... */ };
class Ival_slider : public Ival_box { /* ... */ };
class Ival_dial : public Ival_box { /* ... */ };
class Flashing_ival_slider : public Ival_slider { /* ... */ };
class Popup_ival_slider : public Ival_slider { /* ... */ };
```

и иерархии реализации для различных сторонних графических систем:

```
class BB_ival_slider : public Ival_slider, protected BBslider { /* ... */ };
class BB_flashing_ival_slider : public Flashing_ival_slider,
        protected BBwindow_with_bells_and_whistles { /* ... */ };
class BB_popup_ival_slider : public Popup_ival_slider, protected BBslider { /* ... */
};
class CW_ival_slider : public Ival_slider, protected CWslider { /* ... */ };
// ...
```

Используя очевидные аббревиатуры, эту иерархию можно представить в следующем графическом виде:



Исходная иерархия классов ***Ival_box*** не изменилась — просто она теперь окружена классами графических реализаций.

12.4.3.1. Критика

Дизайн на основе абстрактных классов очень гибок, и к тому же он не менее прост, что и традиционный дизайн на основе общего базового класса, определяющего систему пользовательского интерфейса. В последнем случае класс окон находится в корне дерева. А при первом подходе исходная иерархия классов нашего приложения неизменная и выступает в качестве корневой для классов реализации. С точки зрения работы программы, оба подхода эквивалентны в том смысле, что почти весь код выполняется одинаково. В обоих случаях можно большую часть времени оперировать семейством классов ***Ival_box*** без оглядки на системы пользовательского интерфейса. Например, нам не нужно переписывать ***interact***() из §12.4.1 при смене системы пользовательского интерфейса.

Ясно, что если изменится открытый интерфейс систем пользовательского интерфейса, то придется переписывать реализацию каждого класса из семейства ***Ival_box***. Но в иерархии с абстрактными класса почти весь наш собственный код защищен от изменений в реализации систем пользовательского интерфейса и даже не требует перекомпиляции в таких случаях. Это особенно ценно, когда разработчик систем пользовательского интерфейса выпускает все новые и «почти совместимые» версии. Наконец, пользователи системы с абстрактными классами избавлены от возможности попасть в зависимость от конкретной реализации. Пользователи системы абстрактных классов ***Ival_box*** не могут случайно воспользоваться механизмами конкретных реализаций, так как им доступны лишь средства, явно предоставляемые в иерархии ***Ival_box***, и ничто не наследуется неявным образом от базового класса, специфичного для конкретных систем реализации.

12.4.4. Локализация создания объектов

Большая часть приложения создается на основе интерфейса ***Ival_box***. Далее, если система интерфейсов эволюционирует, предоставляя новые средства, то большая часть приложения использует ***Ival_box***, ***Ival_slider*** и другие интерфейсы. Однако создание объектов требует обращения к конкретным именам, специфичным для систем пользовательского интерфейса — ***CW_ival_dial***, ***BB_flashing_ival_slider*** и т.д. Полезно минимизировать число мест в программе, где создаются объекты. Но создание объектов трудно локализовать, если не делать это систематически.

Как всегда решение находится с помощью применения косвенных обращений. Это можно сделать несколькими способами. Самый простой — предоставить абстрактный класс с набором операций создания объектов:

```
class Ival_maker
{
public:
  virtual Ival_dial* dial(int, int) =0;                    // создать dial
  virtul Popup_ival_slider* popup_slider(int,int) =0;      // создать popup slider
  //...
};
```

Для каждого интерфейса из семейства ***Ival_box***, который знаком пользователю, класс ***Ival_***maker предоставляет функцию создания объекта. Такой класс называют

фабрикой (*factory*), а его функции иногда называют (что может только запутать) виртуальными конструкторами (§15.6.2).

Теперь мы представим каждую систему пользовательского интерфейса классом, производным от ***Ival_maker***:

```
class BB_maker : public Ival_maker    // BB-версия
{
public:
  Ival_dial* dial(int, int);
  Popup_ival_slider* popup_slider(int, int);
  // ...
};

class LS_maker : public Ival_maker    // LS-версия
{
public:
  Ival_dial* dial(int, int);
  Popup_ival_slider* popup_slider(int, int);
  // ...
};
```

Каждая функция создает объект с требуемым интерфейсом и нужной реализацией. Например:

```
Ival_dial* BB_maker::dial(int a, int b)
{
  return new BB_ival_dial(a,b);
}
Ival_dial* LS_maker::dial(int a, int b)
{
  return new LS_ival_dial(a,b);
}
```

Получив указатель на ***Ival_maker***, пользователь может создавать объекты, даже не зная, какая система пользовательского интерфейса задействована. Например:

```
void user(Ival_maker* pim)
{
  Ival_box* pb = pim->dial(0,99);    // создаем подходящий dial
  // ...
}

BB_maker BB_impl;                    // для пользователей BB
LS_maker LS_impl;                    // для пользователей LS

void driver()
{
  user(&BB_impl);                    // используем BB
  user(&LS_impl);                    // используем LS
}
```

12.5. Классовые иерархии и абстрактные классы

Абстрактный класс является интерфейсом. Классовые иерархии служат средством постепенного и последовательного развертывания классов. Естественно, каждый класс формирует интерфейс для своих пользователей, и некоторые абстрактные классы предоставляют определенную полезную функциональность, но тем не менее, главное предназначение абстрактных классов и иерархий состоит в том, чтобы служить «интерфейсами» и «строительными блоками».

Классическая иерархия — это иерархия, в которой отдельные классы предоставляют пользователям полезную функциональность и служат строительными блоками для реализации более специализированных классов. Такие иерархии идеальны для построения программ путем последовательных улучшений. Они обеспечивают максимум поддержки для реализации новых классов до тех пор, пока новые классы точно вписываются в существующую иерархию.

Классические иерархии имеют тенденцию смешивать вопросы реализации с предоставляемыми пользователям интерфейсами. Здесь могут помочь абстрактные классы. Иерархии абстрактных классов предоставляют ясный и мощный способ выражения абстрактных концепций, не загрязняя их деталями реализации и не требуя дополнительных накладных расходов. В конце концов, вызов виртуальной функции обходится дешево и не зависит от того, к какой абстракции идет при этом обращение. Вызов члена абстрактного класса стоит не дороже вызова любой другой *виртуальной* функции.

Логическим завершением этих размышлений будет система, представленная пользователям как иерархия абстрактных классов и реализованная при помощи классической иерархии.

12.6. Советы

1. Избегайте полей типа; §12.2.5.
2. Используйте указатели и ссылки, чтобы избежать срезки; §12.2.3.
3. Используйте абстрактные классы, чтобы сфокусироваться на предоставлении ясных интерфейсов; §12.3.
4. Используйте абстрактные классы для минимизации интерфейсов; §12.4.2.
5. Используйте абстрактные классы для отделения деталей реализации от интерфейсов; §12.4.2.
6. Используйте виртуальные функции, чтобы обеспечить независимость пользовательского кода от добавления новых реализаций; §12.4.1.
7. Используйте абстрактные классы для минимизации необходимых перекомпиляций пользовательского кода; §12.4.2.
8. Используйте абстрактные классы для того, чтобы обеспечить сосуществование альтернативных реализаций; §12.4.3.
9. Класс с виртуальными функциями должен иметь виртуальный деструктор; §12.4.2.

10. Абстрактный класс в типичном случае не нуждается в конструкторах; §12.4.2.
11. Представления различных концепций должны содержаться раздельно; §12.4.1.1.

12.7. Упражнения

1. (*1) Пусть дано определение

```
class base
{
public:
  virtual void iam () {cout<< "base\n" ; }
};
```

   Напишите два производных от ***Base*** класса, и для каждого из них определите ***iam***(), выводящую имя класса. Создайте объекты этих классов и вызовите ***iam***() для них. Присвойте значения указателей на эти объекты указателю типа ***Base**** и вызове с его помощью ***iam***().

2. (*3.5). Реализуйте простую графическую систему, используя доступные на вашем компьютере графические средства (если их нет — используйте ASCII-представление, где пиксел, это знакоместо): ***Window***(***n***, ***m***) создает на экране область размером ***nxm***. Координаты декартовы. Окно ***w*** типа ***Window*** имеет координаты ***w.current***(). Начальные координаты равны ***Point***(***0***, ***0***). Координаты можно задать с помощью ***w.current***(***p***), где ***p*** имеет тип ***Point***. Тип ***Point*** задается парой координат: ***Point***(***x***, ***y***). Тип ***Line*** специфицируется парой точек: ***Line***(***w.current***() , ***p2***); класс ***Shape*** является общим интерфейсом для ***Dot***, ***Line***, ***Rectangle***, ***Circle*** и других фигур. ***Point*** не является ***Shape***. ***Dot***(***p***) представляет точку ***p*** на экране. ***Shape*** на экране не наблюдается до вызова ***draw***(). Например: ***w.draw***(***Circle***(***w.current***() , ***10***)). Каждый ***Shape*** имеет девять контактных точек: ***e*** (***east***), ***w*** (***west***), ***n*** (***north***), ***s*** (***south***), ***ne***, ***nw***, ***se***, ***sw*** и ***c*** (***center***). Например: ***Line***(***x.c***() , ***y.nw***()) создает линию из центра ***x*** в левый верхний угол ***y***. После вызова ***draw***() для ***Shape*** его текущие координаты равны ***se***(). ***Rectangle*** определяется верхней левой и правой нижней вершинами: ***Rectangle***(***w.current***() , ***Point***(***10***, ***10***)). В качестве простого теста изобразите простой детский рисунок «дом с крышей, два окна и дверь».

3. (*2) Изображение ***Shape*** на экране компьютера выполняется в виде множества линейных сегментов. Реализуйте операции, варьирующие внешний вид этих сегментов: ***s.thickness***(***n***) устанавливает толщину линии в ***0***, ***1***, ***2*** или ***3*** единицы, где ***2*** соответствует умолчательному значению, а ***0*** — невидимой линии. Кроме того, сегменты могут быть ***solid*** (сплошная линия), ***dashed*** (пунктирная) и ***dotted*** (из точек), что устанавливается функцией ***Shape::outline***().

4. (*2.5) Предоставьте функцию ***Line::arrowhead***(), добавляющую стрелку к концу линии. У линии два конца и стрелки могут указывать оба направления вдоль линии, так что аргументы функции ***arrowhead***() должны уметь задавать по крайней мере четыре альтернативы.

5. (*3.5) Предоставьте гарантии того, что точки и линии, не попадающие в пределы ***Window***, не появятся на экране компьютера. Это часто называют «отсечением» (clipping). В качестве упражнения не пользуйтесь встроенными средствами вашей графической подсистемы.
6. (*2.5) Добавьте тип ***Text*** к вашей графической системе. ***Text*** — это прямоугольный ***Shape***, внутри которого выводятся символы. По умолчанию, каждый символ занимает позицию в одну единицу по вертикали и горизонтали.
7. (*2) Определите функцию, которая рисует соединяющую две фигуры (типа ***Shape***) линию, вычисляя для этого две «ближайшие точки» и соединяя их.
8. (*3) Добавьте к вашей простой графической системе цвет. Цвет может быть у фона, внутренних областей замкнутых фигур и у контуров фигур.
9. (*2). Рассмотрим:

```
class Char_vec
{
  int sz;
  char element[1];

public:
  static Char_vec* new_char_vec(int s);
  char& operator[](int i) { return element[i]; }
  // ...
};
```

Определите ***new_char_vec***(), выделяющую непрерывную память для ***Char_vec*** таким образом, чтобы доступ к элементам мог осуществляться по индексу через ***element***. При каких обстоятельствах этот трюк может вызвать серьезные проблемы?

10. (*2.5) Для классов ***Circle***, ***Square*** и ***Triangle***, производных от ***Shape***, определите функцию ***intersect***() (пересечение), которая принимает два аргумента типа ***Shape****** и вызывает другие функции, необходимые для выявления возможности пересечения фигур. Для решения этой задачи добавьте к классам соответствующие виртуальные функции. Можете не писать реальный код, выявляющий пересечение: просто убедитесь, что вызываются правильные функции. Решение таких задач называют двойной *диспетчеризацией* (*double dispatch*) или *мультиметодом* (*multi-method*).
11. (*5) Спроектируйте и реализуйте библиотеку для решения задач моделирования, управляемых событиями. Подсказка: <***task.h***>. Это, однако, старая программа, которую вы можете улучшить. Должен быть объявлен класс ***task***, объекты которого могут сохранять состояние и восстанавливать его (функции ***task::save***() и ***task::restore***()), так что они могут работать как индивидуальные задачи. Частные задачи определяются с помощью классов, производных от ***task***. Задачи выполняют программы, специфицируемые их виртуальными функциями. Нужно реализовать возможность передачи параметров задаче с помощью аргументов конструкторов. Должен иметься планировщик (scheduler) для реализации концепции виртуального времени. Введите функцию ***task::delay***(***long***), которая «потребляет» виртуальное время. Сделать планировщик частью класса ***task*** или нет — одно из ваших собственных важ-

нейших проектных решений. Задачи должны иметь возможность взаимодействовать друг с другом (communicate). Разработайте для этого класс ***queue*** (очередь сообщений). Придумайте, как задача могла бы ожидать ввод из разных очередей. Обрабатывайте ошибки времени выполнения единым образом. Как можно отлаживать программы, использующие такую библиотеку?

12. (*2) Определите интерфейсы для ***Warrior*** (воин), ***Monster*** (монстр) и ***Object*** (некий предмет, который можно поднять, бросить и т.д.), применяемых в ролевой игре.
13. (*1.5) Почему одновременно есть и класс ***Point***, и класс ***Dot*** в §12.7[2]? При каких обстоятельствах будет хорошей идеей снабдить ***Shape*** конкретными версиями ключевых классов вроде ***Line***?
14. (*3) Определите в общих чертах конкретные стратегии реализации примера ***Ival_box*** (§12.4), основываясь на идее о том, что каждый видимый приложению класс является интерфейсом, содержащим единственный указатель на реализацию. Таким образом, каждый интерфейсный класс будет выполнять роль дескриптора для класса реализации, и будут существовать иерархии интерфейсов и иерархии реализации. Напишите фрагменты кода, иллюстрирующие возможные проблемы с приведением типов. Рассмотрите возможности упрощения использования, программирования, повторного применения интерфейсов и реализации при добавлении новых концепций к иерархии, упрощение модификации интерфейсов и реализации, а также необходимость перекомпиляции после внесения изменений в реализацию.

13

Шаблоны

Здесь — Ваша цитата.
— Б. Страуструп

Шаблоны — шаблон строк — конкретизация шаблона — параметры шаблонов — проверка типа — шаблоны функций — выведение типов аргументов шаблона — задание аргументов шаблона — перегрузка шаблонов функций — выбор алгоритма через аргументы шаблона — аргументы шаблона по умолчанию — специализация — наследование и шаблоны — шаблонные члены шаблонов — преобразования — организация исходного кода — советы — упражнения.

13.1. Введение

Независимые концепции должны представляться независимо и комбинироваться лишь при необходимости. Когда этот принцип нарушается, вы либо связываете воедино разнородные концепции, либо создаете ненужные зависимости. В любом случае вы порождаете негибкие конструкции для построения программных систем. Шаблоны обеспечивают простой способ представления широкого набора общих концепций и простые способы их комбинирования. Получающиеся результирующие классы и функции не уступают ручным специализированным вариантам по эффективности и потребляемой памяти.

Шаблоны обеспечивают непосредственную поддержку *обобщенного программирования* (*generic programming*) (§2.7), то есть *программирования с использованием типов в качестве параметров*. Механизм шаблонов языка C++ позволяет использовать типы в качестве параметров определений классов или функций. Шаблон зависит лишь от тех свойств параметров-типов, которые он использует явно, и не требует никаких дополнительных зависимостей между ними. В частности, не требуется, чтобы разные параметры-типы шаблона принадлежали к одной иерархии наследования.

В настоящей главе изучение шаблонов фокусируется на тех вопросах, которые заложены в дизайн и реализацию стандартной библиотеки, а также на вопросах ее

практического использования. Стандартная библиотека требует существенно большей степени общности, гибкости и эффективности, чем большинство иных программных систем. Следовательно, использованные в ней технологии действенны и эффективны при разработке программных решений для широкого круга задач и предметных областей. Эти технологии позволяют разработчику скрывать сложные технические детали за простыми интерфейсами и открывать их пользователю лишь тогда, когда в них возникает действительная нужда. Например, через функциональный интерфейс sort(v) открывается доступ ко множеству алгоритмов, сортирующих наборы элементов разного типа, содержащихся в самых разных контейнерах. Наиболее подходящая для конкретного v функция сортировки будет выбрана автоматически.

Любая существенная абстракция стандартной библиотеки представлена шаблоном (например, ***string***, ***ostream***, ***complex***, ***list*** и ***map***), равно как и фундаментальные операции над ними (сравнение строк, операция вывода <<, сложение комплексных чисел, получение следующего элемента списка, сортировка). Специально посвященная стандартной библиотеке часть III настоящей книги является обильным источником по «настоящим» шаблонам и практическим методам работы с ними. А текущая глава опирается на небольшие примеры, которые иллюстрируют следующие технические аспекты шаблонов и фундаментальные способы их использования:

- §13.2: Основные механизмы определения шаблонов классов и их использование.
- §13.3: Шаблоны функций, перегрузка и логический вывод типа.
- §13.4: Параметры шаблонов, управляющие поведением обобщенных алгоритмов.
- §13.5: Множественные определения, обеспечивающие альтернативные реализации шаблонов.
- §13.6: Наследование и шаблоны (полиморфизм на стадии выполнения и полиморфизм на стадии компиляции).
- §13.7: Организация исходного кода.

Начальное знакомство с шаблонами состоялось в §2.7.1 и §3.8. Детальные правила разрешения имен шаблонов, синтаксис шаблонов и т.д. представлены в §C.13.

13.2. Простой шаблон строк

Рассмотрим строку символов. Строка является классом, хранящим символы и обеспечивающим доступ по индексу, конкатенацию, сравнение и другие операции, которые мы обычно ассоциируем с понятием «строка». Такое поведение желательно реализовать для разных наборов символов, например, европейского набора символов, набора китайских символов, греческих и т.д. Поэтому целесообразно определить суть понятия «строка» в отрыве от специфических символьных наборов. Это определение базируется в основном на идее о том, что символы можно копировать. Таким образом, общий строковый тип можно создать из строк символов типа char (§11.12), сделав тип символов параметром:

```
template<class C> class String
{
  struct Srep;
  Srep* rep;

public:
  String();
  String(const C*);
  String(const String&);

  C read(int i) const;
  // ...
};
```

Префикс ***template<class C>*** указывает, что объявляется шаблон, и что идентификатор C является параметром типа. В теле определения шаблона C используется точно так же, как и другие имена типов. Область видимости C простирается до конца определения шаблона. Обратите внимание на то, что ***template<class C>*** означает, что C — это любой *тип*, не обязательно *класс*.

В соответствии с определением шаблона его имя с последующим конкретным типом, заключенным в угловые скобки, является именем класса и его можно использовать так же, как имя любого другого класса. Например:

```
String<char> cs;
String<unsigned char> us;
String<wchar_t> ws;

class Jchar
{
  // японские символы
};

String<Jchar> js;
```

За исключением специального синтаксиса имени, ***String<char>*** ведет себя так же, как и определенный в §11.12 класс ***String***. Преобразовав ***String*** в шаблон, мы получили возможность использовать средства, предназначенные для работы со строками символов типа ***char***, в работе со строками символов иных типов. Например, применив строковый шаблон и стандартный библиотечный контейнер ***map***, мы превратим наш пример с подсчетом слов из §11.8 в следующий код:

```
int main()              // подсчет кол-ва вхождений каждого слова
{
  String<char> buf;
  map<String<char>, int> m;

  while (cin>>buf) m[buf]++;
  // вывод результата
}
```

А вариант для японских символов типа ***Jchar*** будет выглядеть так:

```
int main ()             // подсчет кол-ва вхождений каждого слова
{
  String<char> buf;
  map<String<Jchar>, int> m;

  while (cin>>buf) m [buf] ++;
  // вывод результата
}
```

В стандартной библиотеке определяется шаблонный класс ***basic_string***, аналогичный шаблонному варианту ***String*** (§11.12, §20.3). А ***string*** определяется в стандартной библиотеке как синоним для ***basic_string<char>***:

```
typedef basic_string<char> string;
```

Это позволяет записать программу подсчета слов в следующем виде:

```
int main ()             //подсчет кол-ва вхождений каждого слова
{
  string buf;
  map<string, int> m;

  while (cin>>buf) m [buf] ++;
  // вывод результата
}
```

Вообще, ***typedef*** часто применяют для сокращения длинных имен классов, сгенерированных из шаблонов. Поскольку мы часто предпочитаем не знать всех деталей определения типа, ***typedef*** помогает скрыть тот факт, что тип сгенерирован из шаблона.

13.2.1. Определение шаблона

Класс, генерируемый из классового шаблона, является абсолютно нормальным классом. Применение шаблонов не требует никаких дополнительных механизмов поддержки времени выполнения по сравнению с применением вручную запрограммированных классов. Не предполагает механизм шаблонов и какого-либо уменьшения объема сгенерированного кода.

Обычно хорошей идеей является тестирование и отладка конкретного класса, например ***String***, до его преобразования в шаблон (в нашем примере — в ***String<C>***). Таким образом удается разрешить многие проблемы проектирования и выловить ошибки реализации, оставаясь в рамках конкретного класса. Такой способ отладки знаком всем программистам, а многим из них легче работать с конкретным примером, чем с абстрактной концепцией. Позднее, мы можем сосредоточиться на проблемах, связанных с обобщением, не отвлекаясь при этом на борьбу с обычными ошибками. Аналогично, разбираясь с шаблоном, бывает полезным сначала представить себе его поведение для конкретного типа вместо произвольного типа-параметра, и лишь потом пробовать осознать шаблон во всей его общности.

Члены шаблона класса объявляются и определяются так же, как и в случае обычных классов. Функцию-член шаблона класса не обязательно определять внут-

ри самого шаблона. Ее можно определить где-нибудь еще, как это имеет место и для функций-членов нешаблонных классов (§C.13.7). Члены шаблонных классов сами являются шаблонами, параметризуемыми с помощью параметров шаблона класса. Когда функции-члены определяются вне тела определения шаблона, они должны явным образом указываться как шаблоны. Например:

```
template<class C> struct String<C>::Srep
{
  C* s;                  // указатель на элементы
  int sz;                // число элементов
  int n;                 // подсчет ссылок
  // ...
};

template<class C> C String<C>::read(int i) const {return rep->s[i];}

template<class C> String<C>::String()
{
  rep = new Srep(0,C());
}
```

Параметры шаблонов, такие как C, являются именно параметрами, а не именами типов, определенными внешним образом по отношению к шаблону. Это, однако, никоим образом не влияет на способ, каким мы его используем при написании кода шаблона. В области видимости ***String***<***C***> квалификация с <***C***> избыточна для имени шаблона, так что ***String***<***C***>::***String*** есть имя конструктора. Но если уж мы хотим предельно явных формулировок, то можно писать и так:

```
template<class C> String<C>::String<C>
{
  Rep = new Srep(0,C());
}
```

Аналогично тому, как для обычного класса может существовать лишь одно определение для его функции-члена, также может существовать лишь один шаблон функции в качестве определения функции-члена классового шаблона. Перегрузка возможно только для шаблонов-функций (§13.3.2), в то время как специализация (§13.5) позволяет обеспечить альтернативные реализации шаблонов.

Перегрузка имени шаблона класса невозможна, поэтому в области видимости шаблона никакая другая сущность с тем же именем объявлена быть не может (см. также §13.5). Например:

```
template<class T> class String {/* ... */};
class String{/* ... */}; // error: двойное определение
```

Тип, используемый в качестве аргумента шаблона, должен обеспечивать интерфейс, ожидаемый шаблоном. Например, тип, используемый в шаблоне ***String***, должен предоставлять обычные операции копирования (§10.4.4.1, §20.2.1). Еще раз подчеркнем, что разные типы, используемые в качестве аргументов шаблона, не обязаны находиться между собой в отношениях наследования.

13.2.2. Конкретизация шаблона (template instantiation)

Процесс генерации объявления класса по классовому шаблону и его параметру типа называют *конкретизацией шаблона* (*template instantiation*) (§C.13.7)[1]. Аналогично, конкретная функция генерируется из шаблона функции и заданного значения для параметра типа. Явно определенные программистом версии шаблона для специфических значений параметров типа называются *специализациями* (*specializations*).

Компилятор автоматически генерирует все конкретные версии функций (так что это не является заботой программиста) для любого набора конкретных значений параметров типа (§C.13.7). Например:

```
String<char> cs;

void f()
{
 String<Jchar> js;

 Cs="Какой код сгенерировать, решает компилятор";
}
```

Здесь компилятор генерирует типы ***String<char>***, ***String<Jchar>*** и соответствующие им ***Srep*** типы, код для их деструкторов, умолчательных конструкторов и для операции присваивания ***String<char>::operator=(char*)***. Так как остальные функции-члены здесь не используются, то их код и не генерируется. Сгенерированные классы являются совершенно обычными классами, подчиняющимися всем правилам, имеющим отношение к классам вообще. Аналогично, сгенерированные функции подчиняются всем правилам, относящимся к функциям вообще.

Очевидно, что шаблоны обеспечивают мощный способ генерации кода из относительно небольших определений. Поэтому требуется некоторая осторожность во избежание перегрузки памяти множеством почти идентичных определений функций (§13.5).

13.2.3. Параметры шаблонов

Шаблоны могут иметь параметры типа, параметры обычных фиксированных типов (например, ***int***) и шаблонные параметры (§C.13.3). Естественно, что один шаблон может иметь несколько параметров. Например:

```
template<class T, T def_val> class Cont{/* ... */};
```

Из приведенного примера видно, что параметр типа может использоваться для объявления последующих параметров шаблона.

Аргументы целого типа очень удобны для задания размеров и границ. Например:

```
template<class T, int max> class Buffer
{
  T v[max];
```

[1] Встречающийся в литературе термин *инстанцирование* труден для произношения. — *Прим. ред.*

```
public:
  Buffer() { }
  // ...
};

Buffer<char,128> cbuf;
Buffer<int,5000> ibuf;
Buffer<Record,8> rbuf;
```

Простые, ограниченные контейнеры, вроде ***Buffer***, полезны в задачах, требующих высочайшей эффективности и компактности кода (так что там не подходят универсальные контейнеры типа ***string*** и ***vector***). Передача размера в виде аргумента шаблона ***Buffer*** позволяет избежать затратных операций динамического выделения свободной памяти. Другим примером может послужить тип ***Range*** из §25.6.1.

Аргумент шаблона может быть константным выражением (§C.5), адресом объекта или функции с внешней компоновкой (§9.2), или указателем на член класса (§15.5). Указатель, используемый в качестве аргумента шаблона, должен иметь форму &***of***, где ***of*** — это имя объекта или функции; или же иметь форму ***f***, где ***f*** — это имя функции. Указатель на член класса должен быть в форме &***X***: :***of***, где ***of*** — имя члена. Строковые литералы в качестве аргументов шаблонов не допускаются.

Целый аргумент шаблона должен быть константой:

```
void f(int i)
{
  Buffer<int,i> bx;     // error: требуется константное выражение
}
```

Параметр шаблона, не являющийся типом, в теле шаблона является константой, так что попытка изменить его трактуется как ошибка.

13.2.4. Эквивалентность типов

Располагая шаблоном, мы можем генерировать конкретные типы, предоставляя конкретные значения для аргументов шаблона. Например:

```
String<char> s1;
String<unsigned char> s2;
String<int> s3;

typedef unsigned char Uchar;
String<Uchar> s4;
String<char> s5;

Buffer<String<char>,10> b1;
Buffer<char,10> b2;
Buffer<char,20-10> b3;
```

Предоставляя один и тот же набор аргументов шаблона, мы имеем дело с одним и тем же сгенерированным типом. Но что в данном контексте означает «один и тот же»? Так как typedef не создает нового типа, то ***String<Uchar>*** есть то же самое, что и ***String<unsigned char>***. И наоборот, так как char и unsigned char суть разные типы (§4.3), то и типы ***String<char>*** и ***String<unsigned char>*** тоже разные.

Компилятор может вычислять константные выражения, поэтому тип ***Buffer<char,20-10>*** эквивалентен типу ***Buffer<char,10>***.

13.2.5. Проверка типов

Шаблоны определяются и впоследствии используются в комбинации с набором аргументов шаблона. В определении шаблона можно обнаружить ошибки, не связанные непосредственно с возможными конкретными значениями аргументов шаблона. Например:

```
template<class T> class List
{
  struct Link
  {
    Link* pre;
    Link* suc;
    T val;
    Link (Link* p, Link* s, const T& v) : pre (p) , suc (s) , val (v) { }
  } // syntax error: отсутствует точка с запятой

  Link* head;

public:
  Link () : head (7) { }                           // error: указатель инициализ-ся типом int
  List (const T& t) : head (new Link (0, o, t) ) { } // error: неопределенный идентификатор 'o'
  // ...
  void print_all () const {for (Link* p = head; p; p=p->suc) cout<< p->val<< '\n' ; }
};
```

Компилятор в состоянии распознать простые семантические ошибки в точке определения или позднее в точке использования. Пользователи предпочли бы более раннее выявление ошибок, но не все такие «простые» ошибки легко обнаружить. В примере выше я сделал три «ошибки». Совершенно независимо от того, что есть параметр шаблона, указатель типа ***Link**** не может инициализироваться целым значением *7*. Аналогично, идентификатор о (опечатка — должен был быть нуль) не может быть аргументом конструктора ***List<T>::Link***, поскольку в доступной области видимости такого имени нет.

Любое имя, используемое в определении шаблона, должно либо входить в его область видимости, либо очевидным образом зависеть от параметра шаблона (§С.13.8.1). Наиболее очевидными формами зависимости от параметра шаблона ***T*** является объявление члена типа ***T*** или функции-члена с аргументом типа ***T***. Более тонким примером служит выражение ***cout<<p->val*** в функции-члене ***List<T>::print_all()***.

Ошибки, имеющие отношение к использованию конкретных значений параметров шаблонов, нельзя обнаружить до момента конкретизации шаблона. Например:

```
class Rec { /* ... */ };

void f (const List<int>& li, const List<Rec>& lr)
{
  li.print_all () ;
  lr.print_all () ;
}
```

Здесь выражение ***li.print_all()*** вполне корректно, но выражение ***lr.print_all()*** порождает сообщение об ошибке использования типа, ибо операция вывода << не определена для типа ***Rec***. Самой ранней точкой обнаружения ошибок, связанных

с параметрами шаблонов, является точка первого указания конкретного типа для параметра шаблона. Такую точку программы принято называть первой точкой конкретизации шаблона или просто *точкой конкретизации шаблона* (*point of instantiation*) (см. §C.13.7). Однако в реальных системах разработки (включающих компиляторы) проверки подобного рода могут откладываться до этапа компоновки. Если бы в представленной выше единице трансляции находилось лишь объявление функции ***print_all***(), а не ее определение, то проверка соответствия типов могла быть отложена до более поздних этапов (§13.7). Независимо от того, на каком этапе производится проверка, применяемые для ее выполнения правила одни и те же. Естественно, что для пользователей более ранние проверки предпочтительны. Можно наложить ограничения на допустимые аргументы шаблонов посредством функций-членов (§13.9[16]).

13.3. Шаблоны функций

Для большинства людей самым первым и очевидным применением шаблонов является определение или использование таких контейнеров, как ***basic_string*** (§20.3), ***vector*** (§16.3), ***list*** (§17.2.2) и ***map*** (§17.4.1). Сразу за этим возникает потребность в шаблонных функциях. Сортировка массивов иллюстрирует сказанное:

```
template<class T> void sort(vector<T>&);        // объявление

void f(vector<int>& vi, vector<string>& vs)
{
  sort(vi);    // sort(vector<int>&);
  sort(vs);    // sort(vector<string>&);
}
```

Когда шаблонная функция вызывается, типы фактических аргументов вызова определяют, какая конкретная версия функционального шаблона используется — аргументы шаблона *выводятся* (*deduced*) из типов аргументов функционального вызова (§13.3.1).

Естественно, что шаблон функции должен быть где-то определен (§C.13.7):

```
template<class T> void sort(vector<T>& v)       // определение
                                                // Shell sort (Knuth, Vol. 3, pg. 84).
{
  const size_t n = v.size();
  for(int gap=n/2; 0<gap; gap /= 2)
    for(int i=gap; i<n; i++)
      for(int j=i-gap; 0<=j; j -= gap)
        if(v[j+gap]<v[j])                      // swap v[j] и v[j+gap]
        {
          T temp = v[j];
          v[j]=v[j+gap];
          v[j+gap] = temp;
        }
        else break;
}
```

Сравните это определение с вариантом функции *ssort*() из §7.7. Шаблонная версия яснее и короче, поскольку опирается на типы сортируемых элементов массива. Скорее всего она и более эффективна, поскольку для сравнения элементов не вызывает функцию сравнения по указателю на эту функцию. То есть никаких косвенных вызовов не используется, а встраивание простой операции < весьма несложно.

Для дальнейшего упрощения можно воспользоваться стандартной библиотечной функцией *swap*() (§18.6.8), в результате чего код становится короче и принимает естественный вид:

```
if(v[j+gap]<v[j]) swap(v[j],v[j+gap]);
```

Никаких дополнительных накладных расходов при этом не вносится.

В данном примере для сравнения элементов используется операция <. Поскольку не каждый тип априорно располагает такой операцией, то это ограничивает возможности данной версии функции *sort*(). Однако указанное ограничение легко можно обойти (см. §13.4).

13.3.1. Аргументы функциональных шаблонов

Шаблоны функций чрезвычайно важны для написания обобщенных алгоритмов, применимых к широкому спектру контейнеров (§2.7.2, §3.8, глава 18). Здесь самым существенным моментом является возможность выведения аргументов шаблона по типам аргументов функционального вызова.

Компилятор осуществляет логический вывод аргументов шаблона (параметров типа и иных параметров) при условии, что список аргументов функции однозначно идентифицирует набор аргументов шаблона (§C.13.4). Например:

```
template<class T, int max> T& lookup(Buffer<T,max>& b, const char* p);

class Record
{
  const char v[12];
  //...
};

Record& f(Buffer<Record,128>& buf, const char* p)
{
  return lookup(buf,p);    // вызвать lookup(), где T есть Record, а i - 128
}
```

Для этого примера выводится, что *T* есть *Record*, а *max* равно *128*.

Заметьте, что параметры классовых шаблонов никогда не выводятся. Причина заключается в том, что гибкость, порождаемая наличием нескольких конструкторов класса, в ряде случаев делает вывод параметров шаблона невозможным, а во многих других случаях — неоднозначным. Специализация классовых шаблонов предоставляет возможность неявного выбора между разными реализациями класса (§13.5). Если требуется создать объект выводимого типа, можно просто вызвать функцию, специально предназначенную для такого создания объектов (см., например, *make_pair*() из §17.4.1.2).

Если параметр шаблона не удается вывести из аргументов функционального вызова (§C.13.4), его нужно указать явно. Это делается точно так же, как в случае явного задания параметров для шаблонного класса. Например:

```
template<class T> class vector { /* ... */ };
template<class T> T* create();          // создать T и вернуть указатель на T

void f()
{
  vector<int> v;                        // класс; аргумент шаблона int
  int* p=create<int>();                 // функция; аргумент шаблона int
}
```

Типичным примером явной спецификации типа является необходимость задания конкретного типа возвращаемого шаблонной функцией значения:

```
template<class T, class U> T implicit_cast(U u) {return u; }

void g(int i)
{
  implicit_cast(i);                     // error: невозможно вывести T
  implicit_cast<double>(i);             // T есть double; U есть int
  implicit_cast<char,double>(i);        // T есть char; U есть double
  implicit_cast<char*,int>(i);          // T есть char*; U есть int;
                                        // error: невозможно привести int к char*
}
```

Как и в случае умолчательных значений аргументов обычных функций (§7.5), только концевые аргументы могут опускаться в списке явно задаваемых аргументов шаблона.

Явная спецификация аргументов шаблона позволяет создавать семейства функций преобразования и семейства функций, предназначенных для создания объектов (§13.3.2, §C.13.1, §C.13.5). Часто используются явные версии неявных преобразований (§C.6), такие как ***implicit_cast***(). Синтаксис операций ***dynamic_cast***, ***static_cast*** и т.д. (§6.2.7, §15.4.1) соответствует синтаксису явной квалификации шаблонных функций. Однако некоторые аспекты операций преобразования встроенных типов невыразимы другими средствами языка.

13.3.2. Перегрузка функциональных шаблонов

Можно объявить несколько шаблонов функций с одним и тем же именем, или даже комбинацию шаблонных и обычных функций с совпадающими именами. Когда вызывается функция с перегруженным именем, применяется механизм разрешения перегрузки, позволяющий выбрать наиболее подходящий вариант обычной или шаблонной функции. Например:

```
template<class T> T sqrt(T);
template<class T> complex<T> sqrt(complex<T>);
double sqrt(double);

void f(complex<double> z)
{
  sqrt(2);                              // sqrt<int>(int)
  sqrt(2.0);                            // sqrt(double)
  sqrt(z);                              // sqrt<double>(complex<double>)
}
```

Точно так же, как понятие функционального шаблона является обобщением понятия обычной функции, правила разрешения перегрузки в присутствии функциональных шаблонов являются обобщением правил разрешения перегрузки обычных функций. Сначала отбирается та конкретизация функциональных шаблонов, которая наилучшим образом соответствует фактическим параметрам вызова. Затем она конкурирует с обычными функциями:

1. Отберите конкретизации функциональных шаблонов, которые примут далее участие в процессе разрешения перегрузки. С этой целью рассматривайте каждый функциональный шаблон по отдельности (не обращая внимание на наличие иных шаблонов и обычных функций с тем же именем). Для вызова ***sqrt(z)*** такой отбор в нашем примере порождает двух кандидатов: ***sqrt<double>(complex<double>)*** и ***sqrt<complex<double> >(complex<double>)***.

2. При наличии нескольких кандидатов, порожденных из разных функциональных шаблонов, для дальнейших шагов разрешения перегрузки отбираются наиболее специфические (более узкие, специальные) варианты (§13.5.1). Среди кандидатов, отобранных для нашего примера на шаге [1], предпочтение отдается варианту ***sqrt<double>(complex<double>)*** как более специфическому (любой вызов, удовлетворяющий шаблону ***sqrt<T>(complex<T>)***, удовлетворяет и шаблону ***sqrt<T>(T)***).

3. К отобранным на шаге [2] кандидатам добавляются обычные функции с тем же самым именем и далее применяются правила разрешения перегрузки для обычных функций (§7.4). Если аргумент функционального шаблона был определен в процессе логического вывода (§13.3.1), то к этому аргументу неприменимы «продвижения», стандартные преобразования и пользовательские преобразования. В нашем примере для вызова ***sqrt(2)*** точным соответствием является ***sqrt<int>(int)***, так что он имеет преимущество перед ***sqrt(double)***.

4. Если одинаково хорошо подходят и некоторая конкретизация функционального шаблона и обычная функция, то предпочтение отдается обычной функции. Следовательно, для ***sqrt(2.0)*** предпочтение отдается обычной функции ***sqrt(double)***, а не конкретизации ***sqrt<double>(double)***.

5. Если соответствий не найдено, вызов ошибочен. Если найдены два или более одинаково подходящих варианта, то вызов трактуется как неоднозначный (ambiguous) и тоже ошибочный.

Например:

```
template<class T> T max(T,T);

const int s = 7;

void k()
{
  max(1,2);          // max<int>(1,2)
  max('a','b');      // max<char>('a','b')
  max(2.7,4.9);      // max<double>(2.7,4.9)
  max(s,7);          // max<int>(int(s),7) (тривиальное преобразование)

  max('a',1);        // error: ambiguous (нет стандартного преобразования)
  max(2.7,4);        // error: ambiguous (нет стандартного преобразования)
}
```

Здесь две неоднозначности можно разрешить либо явной квалификацией вызова

```
void f()
{
  max<int>('a',1);          // max(int(int('a'),1)
  max<double>(2.7,4);       // max<double>(2.7,double(4))
}
```

либо добавлением подходящих определений «разрешающих функций»:

```
inline int max(int i, int j) {return max<int>(i,j);}
inline double max(int i, double d) {return max<double>(i,d);}
inline double max(double d, int i) {return max<double>(d,i);}
inline double max(double d1, double d2) {return max<double>(d1,d2);}

void g()
{
  max('a',1);               // max(int('a'),1)
  max(2.7,4);               // max(2.7,4)
}
```

Для обычных функций применяются обычные правила разрешения перегрузки (§7.4), а использование ***inline*** гарантирует отсутствие дополнительных накладных расходов. Разрешающие функции ***max***() столь тривиальны, что можно было бы полностью написать их определения. Однако применение явных конкретизаций функционального шаблона является простым и общим способом определения разрешающих функций.

Сформулированные правила разрешения перегрузки гарантируют корректное взаимодействие функциональных шаблонов и механизма наследования:

```
template<class T> class B { /* ... */ };
template<class T> class D : public B<T> { /* ... */ };
template<class T> void f(B<T>*);

void g(B<int>* pb, D<int>* pd)
{
  f(pb);  //f<int>(pb)
  f(pd);  //f<int>(static_cast<B<int>*>(pd));  // используется стандартное
                                               // преобразование D<int>* в B<int>*
}
```

В этом примере функциональный шаблон ***f***() принимает в качестве аргумента ***B***<***T***>* для любого типа ***T***. Когда фактический аргумент вызова имеет тип ***D***<***int***>*, компилятор легко приходит к выводу, что полагая ***T*** типом ***int***, вызов однозначно разрешается в пользу ***f***(***B***<***int***>*).

Аргумент функции, не используемый в процессе вывода параметра функционального шаблона, трактуется в точности как аргумент обычной (нешаблонной) функции. В частности, к нему применяются обычные правила приведения типов. Рассмотрим пример:

```
template<class T, class C> T get_nth(C& p, int n);   // получить n-ый элемент
```

Эта функция предположительно возвращает значение *n*-го элемента контейнера типа ***C***. Так как ***C*** подлежит логическому выводу из фактического аргумента вызова

функции *get_nth*(), то не допускается преобразований ее первого аргумента. А второй ее аргумент абсолютно обычный, так что по отношению к этому аргументу рассматривается полный набор всех преобразований. Например:

```
class Index
{
public:
  operator int();
  //...
};

void f(vector<int>& v, short s, Index i)
{
  int i1 = get_nth(v,2);     // точное соответствие
  int i2 = get_nth(v,s);     // стандартное преобразование: short в int
  int i3 = get_nth(v,i);     // пользовательское преобразование: Index в int
}
```

13.4. Применение аргументов шаблона для формирования различных вариантов поведения кода

Рассмотрим сортировку строк. Здесь присутствуют три концепции: строка, тип элементов и критерий, используемый сортирующим алгоритмом для сравнения элементов строки.

Мы не можем встроить критерий сортировки непосредственно в контейнер, ибо контейнер не должен навязывать свои проблемы типам элементов. Мы не можем также встраивать критерии и в типы элементов, ибо существует много разных критериев сортировки. Таким образом, критерий сортировки не встраивается ни в контейнер, ни в тип элементов.

Критерий сортировки нужно указывать лишь в тот момент, когда возникла нужда в выполнении конкретной операции. Например, какими критериями сортировки нужно воспользоваться в случае строк, содержащих имена шведов? В этом случае могут используются две разные сортирующие последовательности (два разных способа нумерации символов). Ясно, что ни общий строковый тип, ни общий алгоритм сортировки не должны ничего знать про способы упорядочения шведских имен. Поэтому любое общее решение нуждается в том, чтобы сортирующий алгоритм формулировался в терминах, допускающих настройку не только по типам, но и по вариантам использования этих типов. В качестве примера рассмотрим обобщение стандартной библиотечной функции *strcmp*() для строк элементов типа *T* (§13.2):

```
template<class T, class C>
int compare(const String<T>& str1, const String<T>& str2)
{
  for(int i=0; i<str1.length() && i<str2.length(); i++)
    if(!C::eq(str1[i],str2[i])) return C::lt(str1[i],str2[i]) ? -1 : 1;
```

```
    return str1.length()-str2.length();
}
```

Если потребуется, чтобы ***compare***() игнорировала регистр букв, или действовала в соответствии со специфическими национальными настройками и т.п., то тогда нужно определить подходящий вариант ***C::eq***() и ***C::lt***(). Это позволяет выражать любой алгоритм (сравнение, сортировку и т.п.) в терминах «C-операций» и контейнеров. Например:

```
template<class T> class Cmp   // обычное (умолчательное) сравнение
{
public:
    static int eq(T a, T b) {return a==b;}
    static int lt(T a, T b) {return a<b;}
};

class Literate                // сравнение шведских имен по литературным правилам
{
public:
    static int eq(char a, char b) {return a==b;}
    static int lt(char,char);   // поиск в таблице по коду символа (§13.9[14])
};
```

Теперь мы можем выбирать варианты поведения кода (зависящие в данном случае от правил сравнения строк) явным заданием аргументов шаблона:

```
void f(String<char> swede1, String<char> swede2)
{
    compare<char,Cmp<char> >(swede1,swede2);
    compare<char,Literate>(swede1,swede2);
}
```

Передача операций сравнения в качестве аргумента шаблона имеет два существенных преимущества перед альтернативными решениями, например, перед передачей указателей на функции. Во-первых, можно передать несколько операций в качестве единственного аргумента без дополнительных затрат на выполнение кода. Кроме того, операции сравнения ***eq***() и ***lt***() легко встраиваются, в то время как встраивание вызова функции по указателю на нее является довольно сложной задачей для компилятора.

Естественно, что операции сравнения можно реализовать и для встроенных типов, и для пользовательских типов. Это важно для построения обобщенных алгоритмов, работающих с типами, обладающими нетривиальными критериями сравнения (см. §18.4).

Каждый генерируемый из шаблона класс получает копию каждого *статического* члена классового шаблона (см. §C.13.1).

13.4.1. Параметры шаблонов по умолчанию

Необходимость явного задания критерия сравнения при каждом вызове функции несколько утомительна. По счастью, в общем случае можно опереться на умолчательные варианты критериев, а в редких случаях явно задавать их более специфические варианты. Это можно реализовать с помощью перегрузки:

```
template<class T, class C>
int compare (const String<T>& str1, const String<T>& str2); // сравниваем, используя C

template<class T>
int compare (const String<T>& str1, const String<T>& str2); // сравниваем, используя Cmp<T>
```

По-другому, обычный вариант критерия сравнения можно указать в качестве аргумента шаблона по умолчанию:

```
template<class T, class C = Cmp<T> >
int compare (const String<T>& str1, const String<T>& str2)
{
  for (int i=0; i<str1.length() && i< str2.length(); i++)
    if (!C::eq(str1[i],str2[i])) return C::lt(str1[i],str2[i]) ? -1 : 1;

  return str1.length()-str2.length();
}
```

Теперь можно писать так:

```
void f(String<char> swede1, String<char> swede2)
{
  compare(swede1,swede2);                  // используется Cmp<char>
  compare<char,Literate>(swede1,swede2);   // используется Literate
}
```

Или для менее экзотического (по сравнению со шведскими фамилиями) сравнения с учетом и без учета регистра букв:

```
class No_case { /* ... */ };

void f(String<char> s1, String<char> s2)
{
  compare(s1,s2);                  // учитываем регистр
  compare<char,No_case>(s1,s2);    // не учитываем регистр
}
```

Технология, позволяющая осуществлять задание вариантов поведения кода через аргументы шаблонов с использованием их умолчательных значений для наиболее общих случаев, широко применяется в стандартной библиотеке (§18.4). Достаточно странно, но она не используется для типа ***basic_string*** (§13.2, глава 20). Параметры шаблонов, применяемые для задания вариантов поведения кода, часто называют «свойствами» (traits). Например, строки стандартной библиотеки используют ***char_traits*** (§20.2.1), стандартные алгоритмы полагаются на соответствующие свойства итераторов (§19.2.2), а контейнеры стандартной библиотеки — на *аллокаторы* (*allocators*) (§19.4).

Проверка семантики умолчательного значения параметра шаблона проводится только в случае его использования. Например, пока мы воздерживаемся от использования умолчательного значения ***Cmp<T>***, мы можем использовать функцию ***compare***() для сравнения строк элементов типа ***X***, для которых ***Cmp<X>*** не компилируется (например, потому что операция < не определена для типа ***X***). Это очень важно для стандартных контейнеров, применяющих умолчательные значения аргументов шаблона (§16.3.4).

13.5. Специализация

По умолчанию, шаблон является единственным определением, которое должно использоваться для всех конкретных аргументов шаблона, задаваемых пользователем. Это не всегда оптимально для разработчика шаблона. Для него часто актуально рассуждать так: «если аргумент шаблона будет указателем, нужно применить вот эту реализацию, а если нет — то другую реализацию» или «в случае, когда аргумент шаблона не является указателем на класс, производный от ***My_base***, выдать сообщение об ошибке». Такие проблемы можно решить, обеспечив альтернативные определения шаблонов и заставив компилятор выбирать нужный вариант на основе аргументов шаблона, указанных при его использовании. Такие альтернативные определения шаблона называются *специализациями, определяемыми пользователем* (*user-defined specializations*), или просто *пользовательскими специализациями* (*user specializations*)[1].

Рассмотрим типичные варианты применения шаблона ***Vector***:

```
template<class T> class Vector              // general vector type
{
  T* v;
  int sz;

public:
  Vector();
  explicit Vector(int);

  T& elem(int i) {return v[i];}
  T& operator[](int i);

  void swap(Vector&);
//...
};

Vector<int> vi;
Vector<Shape*> vps;
Vector<string> vs;
Vector<char*> vpc;
Vector<Node*> vpn;
```

Большинство векторов строятся на указателях некоторого типа. На то существует ряд причин, но главная заключается в том, что только указатели обеспечивают полиморфное поведение на этапе выполнения (§2.5.4, §12.2.6). В итоге, любой программист, практикующий объектно-ориентированное программирование и использующий безопасные по отношению к типам контейнеры (такие как контейнеры стандартной библиотеки), так или иначе приходит к контейнерам указателей.

В большинстве реализаций C++ код шаблонов функций реплицируется. Это хорошо с точки зрения производительности, но при недостаточной осторожности может привести к разбуханию кода в критических случаях, как в случае шаблона ***Vector***.

[1] Еще чаще их называют просто *специализациями*. — *Прим. ред.*

По счастью, имеется очевидное решение. Все контейнеры указателей могут разделить единственную специальную реализацию, называемую *специализацией* (*specialization*). Определяем специфическую версию (специализацию) шаблона **Vector** для указателей на **void**:

```
template<> class Vector<void*>
{
  void** p;
  //...
  void*& operator[] (int i);
};
```

Эта специализация может затем использоваться как общая реализация для всех векторов указателей.

Префикс **template**<> говорит о том, что определяется специализация, не нуждающаяся в параметрах шаблона. Аргументы шаблона, для которых эта специализация должна использоваться, указываются в угловых скобках после имени. Таким образом, <**void***> означает, что данное определение должно использоваться в качестве реализации для всех **Vector**, у которых **T** есть **void***.

Определение **Vector**<**void***> называется *полной специализацией* (*complete specialization*), поскольку параметры шаблона отсутствуют. Оно используется в клиентском коде следующим образом:

```
Vector<void*> vpv;
```

Для того чтобы определить специализацию, которую можно использовать для любого вектора указателей (и только для векторов указателей), требуется *частичная специализация* (*partial specialization*):

```
template<class T> class Vector<T*> : private Vector<void*>
{
public:
  typedef Vector<void*> Base;

  Vector() {}
  explicit Vector(int i) : Base(i) {}

  T*& elem(int i) {return reinterpret_cast<T*&>(Base::elem(i)); }
  T*& operator[] (int i) {return reinterpret_cast<T*&>(Base::operator[] (i)); }
//...
};
```

Паттерн (схема, форма) специализации (*specialization pattern*) в виде <**T***> после имени означает, что специализация используется для любого типа указателей; то есть это определение используется всегда, когда указывается аргумент шаблона в виде **T***. Например:

```
Vector<Shape*> vps;   // <T*> есть <Shape*>, так что T есть Shape
Vector<int**> vppi;   // <T*> есть <int**>, так что T есть int*
```

Отметим, что когда используется частичная специализация, параметр шаблона выводится из паттерна специализации, так что в этом случае параметр шаблона не совпадет с фактическим аргументом шаблона. В частности, для **Vector**<**Shape***> параметр **T** есть **Shape**, а не **Shape***.

Определив рассмотренную частичную специализацию для шаблона ***Vector***, мы получили общую реализацию для любых векторов указателей. При этом по сути дела ***Vector***<***T****> является интерфейсом к ***Vector***<***void****>, реализованным через наследование и оптимизируемым за счет встраивания.

Важно, что все эти усовершенствования реализации ***Vector*** выполнены без изменения интерфейса пользователя. Специализация как раз и задумана как альтернативная реализация для специфических случаев использования одного и того же пользовательского интерфейса. Естественно, можно назначить разные имена общему шаблону (вектору вообще) и специализации (вектору указателей), но это будет только путать пользователей, и многие из них могут и не воспользоваться специализацией для векторов указателей, что лишь раздует суммарный объем их кода. Гораздо лучше скрывать важные детали реализации за общим интерфейсом.

Рассмотренная техника программирования, препятствующая разбуханию кода, доказала свою эффективность на практике. Люди, не применяющие этой техники программирования (будь-то в языке C++, или в каком-либо ином языке с аналогичными средствами параметризации), быстро обнаруживают, что излишне продублированный код может запросто потянуть на десятки мегабайт даже в программах весьма умеренного размера. Из-за того, что отсутствует необходимость в компиляции этого избыточного кода, общее время компиляции и компоновки может значительно сократиться.

Применим единственную специализацию для любых списков указателей в качестве еще одного примера минимизации общего кода за счет увеличения доли разделяемого (общего для разных типов) кода. Общий шаблон должен быть объявлен до любой его специализации. Например:

```
template<class T> class List<T*> { /* ... */ };
template<class T> class List { /* ... */ };  // error: общий шаблон после специализации
```

Критически важной информацией в определении общего шаблона является набор параметров шаблона, которые пользователь должен предоставлять, используя этот шаблон или его специализации. Поэтому *одного лишь объявления* общего шаблона достаточно для объявления или определения *специализации*:

```
template<class T> class List;
template<class T> class List<T*> { /* ... */ };
```

Если общий шаблон используется в клиентском коде, он, естественно, должен быть где-то определен (§13.7).

Специализация должна находиться в области видимости кода, в котором пользователь использует эту специализацию. Например:

```
template<class T> class List { /* ... */ };

List<int*> li;

template<class T> class List<T*> { /* ... */ };  // error
```

Здесь ***List*** был специализирован под ***int**** позже того, как с помощью объявляющей конструкции ***List***<***int****> был уже использован.

Все специализации шаблона должны объявляться в том же самом пространстве имен, что и общий шаблон. Если специализация используется, то она должна быть где-то явным образом определена (§13.7). Другими словами, явная специа-

лизация шаблона означает, что автоматическая генерация ее определения не выполняется.

13.5.1. Порядок специализаций

Одна специализация считается более специализированной (более специфической, узкой), чем другая специализация, если список фактических аргументов шаблона, удовлетворяющий ее паттерну специализации, также соответствует и паттерну второй специализации, но не наоборот. Например:

```
template<class T> class Vector;           // общий шаблон
template<class T> class Vector<T*>;       // специализация для любых указателей
template<> class Vector<void*>;           // специализация для void*
```

Любой тип может использоваться как аргумент для самого общего шаблона *Vector*, но только указатели подходят для специализации *Vector<T*>*, и только тип *void** — для специализации *Vector<void*>*.

Более специальным версиям отдается предпочтение в объявлениях объектов, указателей и т.д. (§13.5), а также при разрешении перегрузки (§13.3.2).

Паттерн специализации может быть специфицирован в терминах типов с применением конструкций, допустимых при выводе параметра шаблона (§13.3.1, §C.13.4).

13.5.2. Специализация шаблонов функций

Естественно, что специализация полезна и для шаблонов функций. Рассмотрим сортировку Шелла из §7.7 и §13.3. В ней производится сравнение элементов операцией < и в деталях выполняется перестановка элементов. Предпочтительнее дать следующее определение:

```
template<class T> bool less(T a, T b) {return a<b; }

template<class T> void sort(Vector<T>& v)
{
  const size_t n = v.size();

  for(int gap=n/2; 0<gap; gap /= 2)
    for(int i=gap; i<n; i++)
      for(int j=i-gap; 0<=j; j -= gap)
        if(less(v[j+gap], v[j]))
          swap(v[j], v[j+gap]);
        else
          break;
}
```

Это не улучшает сам алгоритм, а лишь его реализацию.

В том виде, в каком шаблон *sort*() сейчас определен, он не сможет корректно сортировать тип *Vector<char*>*, поскольку операция < будет сравнивать два указателя на *char*, то есть сравнивать адреса первых символов каждой строки. Нам же нужно, чтобы сравнивались сами символы. Простая специализация функционального шаблона *less*() для типа *const char** позаботится об этом:

```
template<> bool less<const char*> (const char* a, const char* b)
{
  return strcmp (a,b) <0;
}
```

Как и для классовых шаблонов (§13.5), для функциональных шаблонов префикс ***template***<> означает специализацию, не нуждающуюся в параметрах шаблона. Конструкция <***const char****> после имени функции означает, что эта специализация должна использоваться в случае, когда фактические параметры вызова имеют тип ***const char****. Поскольку параметры функциональных шаблонов могут быть выведены из фактических параметров вызова, нет нужды специфицировать их явно. Поэтому мы можем упростить определение специализации:

```
template<> bool less<> (const char* a, const char* b)
{
  return strcmp (a,b) <0;
}
```

При наличии префикса ***template***<> присутствие вторых пустых угловых скобок излише, поэтому лучше писать так:

```
template<> bool less (const char* a, const char* b)
{
  return strcmp (a,b) <0;
}
```

Во многих случаях перегрузка (§13.3.2) является альтернативой специализации. Рассмотрим очевидное определение для ***swap***():

```
template<class T> void swap (T& x, T& y)
{
  T t = x;  // копируем x во временную переменную
  x = y;    // копируем y в x
  y = t;    // копируем временную переменную в y
}
```

Такое решение неэффективно для типа ***Vector***, поскольку при этом будут копироваться все элементы векторов. Элементы x будут даже копироваться дважды. Проблема решается предоставлением версии ***swap***(), более подходящей для векторов. Объект типа ***Vector*** будет представлен только данными, достаточными для косвенного доступа к элементам (как ***String***; §11.12, §13.2). Таким образом, работу ***swap***() можно свести к обмену этими представлениями. Для этого я определяю ***swap***() как функцию-член шаблона ***Vector*** (§13.5):

```
template<class T> void Vector<T>::swap (Vector& a)     // меняемся представлениями
{
  swap (v,a.v);
  swap (sz,a.sz);
}
```

Теперь эту функцию-член можно использовать в качестве альтернативы общему шаблону ***swap***():

```
template<class T> void swap (Vector<T>& a, Vector<T>& b)
{
  a.swap (b) ;
}
```

Рассмотренные здесь специализация *less*() и перегруженная версия *swap*() используются в стандартной библиотеке (§16.3.9, §20.3.16). Кроме того, они отлично иллюстрируют широко применяемые приемы программирования. Специализация и перегрузка полезны тогда, когда имеется более эффективная альтернатива общему алгоритму для конкретных аргументов шаблона (здесь это *swap*()). Кроме того, специализация отлично работает тогда, когда «иррегулярность» типа аргумента приводит к неправильной работе общего алгоритма (здесь это *less*()). Часто такими «иррегулярными» типами являются встроенные типы указателей и массивов.

13.6. Наследование и шаблоны

Шаблоны и наследование являются механизмами построения новых типов из уже существующих, а также помогают писать код, извлекающий пользу из самых разных форм общности. Как показано в §3.7.1, §3.8.5 и §13.5, комбинация этих двух механизмов является основой для многих полезных технологий программирования.

Наследование классового шаблона от нешаблонного класса позволяет реализовать общую реализацию для множества шаблонов. Вектор из §13.5 служит хорошим примером:

```
template<class T> class Vector<T*> : private Vector<void*> { /* ... */ };
```

Под другим углом зрения этот пример иллюстрирует тот факт, что шаблон используется для предоставления элегантного и типобезопасного интерфейса к средству, в противном случае являющегося и опасным, и неудобным.

Естественно, часто бывает полезным создавать классовые шаблоны, производные от других классовых шаблонов. Базовый класс, в частности, служит строительным блоком в реализации производных классов. Если члены базового класса должны соответствовать тому же параметру, что и члены производного класса, то имя базового класса следует соответственно параметризовать; хорошим примером служит *Vec* из §3.7.2:

```
template<class T> class vector { /* ... */ };
template<class T> class Vec : public vector<T> { /* ... */ };
```

Правила разрешения перегрузки шаблонных функций гарантируют, что функции работают правильно для таких производных типов (§13.3.2).

Случай, когда производный и базовый классы имеют один и тот же параметр шаблона, является наиболее распространенным, но это не обязательное требование. В редких и интересных случаях в качестве параметра базового класса указывается сам производный класс. Например:

```
template<class C> class Basic_ops      // базовые операции над контейнерами
{
public:
  bool operator== (const C&) const;    // сравнивает все элементы
```

```
    bool operator!=(const C&) const;
    // ...
    // доступ к операциям типа C:
    const C& derived() const {return static_cast<const C&>(*this);}
};

template<class T> class Math_container : public Basic_ops<Math_container<T> >
{
public:
    size_t size() const;
    T& operator[](size_t);
    const T& operator[](size_t) const;
    // ...
};
```

Это позволяет один раз определить базовые операции над контейнерами и отделить их от определений самих контейнеров. В то же время, поскольку определение операций вроде == и != должно выражаться в терминах и контейнера, и его элементов, то тип элементов должен передаваться шаблону контейнера.

Полагая, что ***Math_container*** аналогичен традиционному вектору, определения членов ***Basic_ops*** будут выглядеть примерно так:

```
template<class C> bool Basic_ops<C>::operator==(const C& a) const
{
    if(derived().size != a.size()) return false;

    for(int i=0; i<derived().size(); ++i)
        if(derived()[i] != a[i]) return false;

    return true;
}
```

Другой техникой разделения контейнеров и операций является их комбинирование через параметры шаблонов (а не использование наследования):

```
template<class T, class C> class Mcontainer
{
    C elements;

public:
    T& operator[](size_t i) {return elements[i];}

    friend bool operator==<>const Mcontainer&, const Mcontainer&);
    friend bool operator!=<>(const Mcontainer&, const Mcontainer&);
    // ...
};

template<class T> class My_array { /* ... */ };
Mcontainer<double,My_array<double> > mc;
```

Класс, генерируемый из шаблона класса, является обычным классом. Следовательно, он может иметь дружественные функции (§C.13.2). Я в данном примере воспользовался дружественными функциями, чтобы выразить симметрию операций == и != (§11.3.2).

Можно было бы рассмотреть и вариант с передачей шаблона (шаблонный параметр классового шаблона) вместо контейнера для аргумента *C* (§C.13.3).

13.6.1. Параметризация и наследование

Шаблоны являются механизмом параметризации определения типа или функции другим типом. Исходный код, реализующий шаблон, одинаков для всех типов параметров, как и большая часть кода, использующего шаблон. Когда требуется дополнительная гибкость, определяют специализации. Абстрактный класс определяет интерфейс. Большая часть кода различных реализаций абстрактного класса может совместно использоваться в классовой иерархии, и большая часть кода, использующего абстрактный класс, не зависит от его реализации. С точки зрения проектирования оба подхода настолько близки, что заслуживают общего названия. Так как оба позволяют выразить алгоритм единожды и использовать его со многими типами, люди все это называют *полиморфизмом*. Чтобы все-таки различать эти подходы, для виртуальных функций применяют термин *полиморфизм времени выполнения* (*run-time polymorphism*), а для шаблонов предлагают термин *полиморфизм времени компиляции* (*compile-time polymorphism*) или *параметрический полиморфизм* (*parametric polymorphism*).

Итак, когда же следует выбирать шаблоны, а когда — абстрактные классы? В обоих случаях мы манипулируем объектами, которые разделяют (совместно используют) общий набор операций. Если отсутствуют иерархические взаимозависимости между объектами, лучше выбирать шаблоны. Когда же истинный тип объектов не известен на этапе компиляции, нужно опираться на классовые иерархии наследования от общего абстрактного класса. Когда же особо важна производительность, обеспечиваемая за счет встраивания операций, следует использовать шаблоны. Эти вопросы подробнее обсуждаются в §24.4.1.

13.6.2. Шаблонные члены шаблонов

Класс или классовый шаблон могут иметь члены, которые сами являются шаблонами. Например:

```
template<class Scalar> class complex    // детали см. в §22.5
{
  Scalar re, im;

public:
  template<class T> complex(const complex<T>& c) : re(c.real()),im(c.imag()) {}
  //...
};

complex<float> cf(0,0);
complex<double> cd = cf;          // ok: используется приведение float к double

class Quad
{
  // отсутствует приведение к int
};

complex<Quad> cq;
complex<int> ci=cq;               // error: нет приведения Quad к int
```

Другими словами, мы можете создавать ***Complex<T1>*** из ***Complex<T2>*** тогда и только тогда, когда вы можете инициализировать ***T1*** с помощью ***T2***. Это выглядит разумно.

К сожалению, C++ допускает некоторые неразумные операции преобразования встроенных типов, такие как ***double*** в ***int***. Сопутствующие проблемы потери точности можно обнаружить во время выполнения с помощью проверяемых преобразований в стиле ***implicit_cast*** (§13.3.1) и функции ***checked*** () (§C.6.2.6):

```
template<class Scalar> class complex    // детали см. в §22.5
{
  Scalar re, im;

public:
  complex () : re (0) , im (0) { }
  complex (const complex<Scalar>& c) : re (c.real () ) , im (c.imag () ) { }

  template<class T2> complex (const complex<T2>& c)
             :
re (checked_cast<Scalar> (c.real () ) ) , im (checked_cast<Scalar> (c.imag () ) ) { }

  // ...
};
```

Для полноты картины я добавил умолчательный конструктор и копирующий конструктор. Любопытно, что шаблонный конструктор (в этом примере — зависящий от параметра шаблона ***T2***) никогда не используется для генерации обычного копирующего конструктора, так что в отсутствие явного определения последнего будет сгенерирован умолчательный вариант копирующего конструктора. В нашем случае такой неявно сгенерированный копирующий конструктор был бы идентичен тому, что я определил явно. Аналогично, операция присваивания (§10.4.4.1, §11.7) также не должна определяться как шаблонная функция-член.

Шаблонная функция-член не может быть виртуальной. Например:

```
class Shape
{
  // ...
  template<class T> virtual bool intersect (const T&) const=0;    // error: virtual template
};
```

Такое недопустимо. Если бы это было позволено, то для реализации виртуальных функций нельзя было бы применить традиционную технику виртуальных таблиц (§2.5.5). Компоновщику пришлось бы добавлять новую точку входа в виртуальную таблицу для класса ***Shape*** каждый раз, когда ***intersect*** () вызывается с новым типом аргумента.

13.6.3. Отношения наследования

Шаблон класса полезно представлять как спецификацию того, как должны создаваться конкретные типы. Другими словами, реализация шаблона является механизмом генерации типов, основанная на предоставляемых пользователем сведениях. Как следствие, шаблоны классов часто называют *генераторами типов* (*type generators*).

С точки зрения языка C++ два типа, сгенерированные из одного шаблона, никак не связаны между собой. Например:

```
class Shape { /* ... */ };
class Circle : public Shape { /* ... */ };
```

Исходя из представленных объявлений люди часто пытаются трактовать *set<Circle*>* как *set<Shape*>*. Это серьезная логическая ошибка, основанная на неверном рассуждении: «*Circle* это *Shape*, так что множество объектов типа *Circle* есть в то же время и множество объектов типа *Shape*; следовательно, я могу использовать множество объектов *Circle* как множество объектов *Shape*». В этом рассуждении последняя часть не верна. Причина состоит в том, что множество объектов *Circle* гарантирует, что каждый член множества есть *Circle*, в то время как множество *Shape* такой гарантии не дает. Например:

```
class Triangle : public Shape { /* ... */ };

void f(set<Shape*>& s)
{
  // ...
  s.insert(new Triangle());
  // ...
}

void g(set<Circle*>& s)
{
  f(s);    // error: несоответствие типов: s есть set<Circle*>, а не set<Shape*>
}
```

Этот код не будет компилироваться, так как нет встроенного преобразования от *set<Circle*>&* к *set<Shape*>&*. И его не должно быть. Гарантия того, что члены множества *set<Circle*>* есть объекты типа *Circle*, позволяет нам безопасно и эффективно применять операции, специфичные для окружностей, например, определение радиуса, ко всем членам множества. Если бы мы позволили с *set<Circle*>* обращаться как с *set<Shape*>*, то мы нарушили бы такую гарантию. Например, функция *f()* внедряет новый треугольник (объект типа *Triangle*) во множество *set<Shape*>*, передаваемое ей в качестве аргумента. Тогда, если бы множество *set<Shape*>* оказалось бы при вызове *f()* множеством *set<Circle*>*, была бы нарушена фундаментальная гарантия того, что все элементы множества *set<Circle*>* есть окружности.

13.6.3.1. Преобразования шаблонов

В примере из предыдущего раздела демонстрируется, что не может существовать отношений по умолчанию между классами, сгенерированными из одного и того же шаблона. В то же время, для некоторых шаблонов нам хотелось бы такие отношения выразить. Например, когда мы определяем шаблон указателей (объекты сгенерированных из шаблона классов ведут себя как указатели), нам хотелось бы отразить отношения наследования между адресуемыми объектами. Шаблонные члены шаблонов (§13.6.2) позволяют выразить некоторые из таких взаимоотношений. Рассмотрим пример:

```
template<class T> class Ptr          // указатель на T
{
  T* p;
```

```
public:
  Ptr(T*);
  Ptr(const Ptr&);                          // копирующий конструктор
  template<class T2> operator Ptr<T2>();    // преобразование Ptr<T> в Ptr<T2>
  //...
};
```

Мы хотим определить операции преобразования для пользовательского типа ***Ptr*** так, чтобы операции преобразования между объектами этого типа были такими же, как между встроенными указателями при наследовании. Например:

```
void f(Ptr<Circle> pc)
{
  Ptr<Shape> ps  = pc;                      // должно работать
  Ptr<Circle> pc2 = ps;                     // даст ошибку
}
```

Мы хотим допускать первую из указанных в данном коде инициализаций в том и только в том случае, когда ***Shape*** является прямым или косвенным открытым базовым классом для ***Circle***. В общем, нам нужно определить операцию преобразования так, чтобы преобразование от ***Ptr<T>*** к ***Ptr<T2>*** допускалось тогда и только тогда, когда значение типа ***T**** можно присваивать объекту типа ***T2****. Этого можно достичь следующим образом:

```
template<class T>
template<class T2>
Ptr<T>::operator Ptr<T2>() {return Ptr<T2>(p);}
```

Здесь оператор return будет компилироваться тогда и только тогда, когда указатель ***p*** (имеет тип ***T****) может быть аргументом конструктора ***Ptr<T2>(T2*)***. Поэтому, если ***T**** может неявно приводиться к ***T2****, то преобразование ***Ptr<T>*** в ***Ptr<T2>*** будет работать. Например:

```
void f(Ptr<Circle> pc)
{
  Ptr<Shape> ps  = pc;                      // ok: можно привести Circle* к Shape*
  Ptr<Circle> pc2 = ps;                     // error: нельзя привести Shape* к Circle*
}
```

Старайтесь определять лишь логически осмысленные операции преобразования.

Обратите внимание на то, что списки параметров самого шаблона и его шаблонных членов объединять нельзя. Например:

```
template<class T, class T2>                 // error
Ptr<T>::operator Ptr<T2>() {return Ptr<T2>(p);}
```

13.7. Организация исходного кода

Существует два очевидных способа организации кода, использующего шаблоны:

1. Включать определения шаблонов до их использования в той же самой единице трансляции.

2. Включать лишь объявления шаблонов до их использования в единице трансляции, а определения компилировать отдельно.

Кроме того, шаблонные функции могут сначала объявляться, затем использоваться и лишь после этого определяться в одной и той же единице трансляции.

Для иллюстрации различий между двумя подходами рассмотрим простой шаблон:

```
#include <iostream>

template<class T> void out (const T& t) {std::cerr << t; }
```

Мы могли бы поместить этот фрагмент в файл ***out.c***, и включать его директивой ***#include*** в те места, где вызывается ***out***(). Например:

```
// user1.c:
#include "out.c"
// используем out()

// user2.c:
#include "out.c"
// используем out()
```

Таким образом, определение функции ***out***() и все объявления, от которых она зависит, должны помещаться в каждую единицу трансляции, где ***out***() используется. Далее уже от компилятора зависит, как именно он будет бороться с избыточными определениями и устранять эту избыточность (по возможности). Такая стратегия трактует шаблонные функции так же, как встраиваемые функции. Ее недостатком является излишняя нагрузка на компилятор, который должен переваривать большие объемы информации. Другая проблема состоит в том, что пользователи могут случайно получить ненужные дополнительные зависимости от объявлений, необходимых лишь определению функционального шаблона ***out***(). Опасность этой проблемы можно минимизировать, используя пространства имен, избегая макросов и, вообще, минимизируя количество дополнительно включаемой информации.

Другим подходом является стратегия раздельной компиляции, которая вытекает из следующего рассуждения: если определение шаблона не включается в код пользователя, то никакие нужные шаблону объявления не влияют на пользовательский код. Таким образом, мы разбиваем наш файл ***out.c*** на два файла:

```
// out.h:
template<class T> void out (const T& t) ;

// out.c:
#include<iostream>
#include "out.h"

export template<class T> void out (const T& t) {std::cerr <<t; }
```

Теперь файл ***out.c*** содержит всю информацию, необходимую для определения ***out***(), а файл ***out.h*** содержит лишь информацию, необходимую для вызова ***out***(). При этом подходе в пользовательский код включается только объявление шаблона (интерфейс):

```
// user1.c:
#include "out.h"
// используем out()

// user2.c:
#include "out.h"
// используем out()
```

Эта стратегия трактует шаблонные функции как обычные невстраиваемые функции. Определение (в файле *out.c*) компилируется отдельно, и задачей реализации является найти его при необходимости. Это также нагружает реализацию, но по-другому, чем в случае борьбы с избыточными определениями.

Отметим, что определение (или объявление) шаблона должно быть дано с ключевым словом *export* (§9.2.3)[1], или оно будет недоступно из клиентского кода. В противном случае оно должно находиться в той области видимости, где оно используется.

Какая стратегия (или их комбинация) является оптимальной, зависит от используемых компилятора и компоновщика, от внутренних особенностей разрабатываемой программной системы, и от внешних ограничений, накладываемых на нее. Обычно, встраиваемые функции и небольшие шаблонные функции, которые главным образом вызывают другие шаблонные функции, являются кандидатами на включение в каждую единицу трансляции, в которой они используются. Для реализаций с умеренной поддержкой компоновщика в вопросе конкретизации шаблонов такой подход ускоряет процесс компиляции и повышает точность сообщений об ошибках.

Включение определений делает их уязвимыми от макросов и объявлений в контексте использования. Следовательно, большие шаблоны и шаблоны со сложными контекстными зависимостями лучше компилировать отдельно. Кроме того, это позволяет пользовательскому коду избавиться от влияния объявлений, требующихся для определения шаблона.

Я считаю подход с включением лишь объявлений шаблонов в пользовательский код и раздельной компиляцией определений шаблонов идеальным. Однако практическое воплощение идеалов часто сдерживается практическими ограничениями, к тому же раздельная компиляция шаблонов на некоторых реализациях является слишком громоздкой и дорогой операцией.

Независимо от того, какая стратегия используется, невстраиваемые статические члены (§C.13.1) должны иметь уникальные определения в некоторых единицах трансляции. Это подразумевает, что такие члены лучше не использовать в шаблонах, если последние предполагается включать во многие единицы трансляции.

Идеальным является код, который одинаково хорошо работает как в случае, когда он компилируется из одной единицы трансляции, или собирается из нескольких независимо компилируемых единиц. К такому идеалу нужно стремиться, ограничивая зависимость определений шаблонов от контекста, и не пытаясь втискивать все это в единый процесс конкретизации.

---

[1] Многие компиляторы (в частности фирмы Microsoft) на платформе Windows не поддерживают это ключевое слово. — *Прим. ред.*

13.8. Советы

1. Используйте шаблоны для представления алгоритмов, применимых к разным типам аргументов; §13.3.
2. Используйте шаблоны для реализации контейнеров; §13.2.
3. Для минимизации размера кода создавайте специализации для контейнеров указателей; §13.5.
4. Всегда объявляйте общую форму шаблона до его специализаций; §13.5.
5. Объявляйте специализацию до ее использования; §13.5.
6. Минимизируйте зависимость определения шаблона от контекста его конкретизации; §13.2.5, §C.13.8.
7. Определяйте все объявленные специализации; §13.5.
8. Подумайте, нужны ли специализации вашего шаблона для строк и массивов языка C; §13.5.2.
9. Параметризуйте шаблоны аргументами, задающими различные варианты поведения; §13.4.
10. Для предоставления единого интерфейса к разным реализациям шаблона для различных типов используйте специализации и перегрузку; §13.5.
11. Предоставляйте простой интерфейс для простых случаев и применяйте перегрузку и аргументы по умолчанию для более специальных случаев; §13.5, §13.4.
12. Отлаживайте конкретные примеры до их обобщения в шаблоны; §13.2.1.
13. Не забывайте про ключевое слово ***export*** в случаях, когда определения шаблонов нужно компоновать с другими единицами трансляции; §13.7.
14. Отдельно компилируйте большие шаблоны и шаблоны с нетривиальными зависимостями от контекста; §13.7.
15. Используйте шаблоны, чтобы выразить преобразования, но делайте это осмотрительно; §13.6.3.1.
16. При необходимости ограничивайте аргументы шаблона с помощью функции-члена ***constraint***(); §13.9[16], §C.13.10.
17. Для минимизации времени компиляции используйте явную конкретизацию; §C.13.10.
18. Когда скорость исполнения имеет первостепенную важность, применяйте шаблоны вместо наследования; §13.6.1.
19. Используйте механизм наследования вместо шаблонов, когда важно иметь возможность добавления новых вариантов без перекомпиляции клиентского кода; §13.6.1.
20. Отдавайте предпочтение шаблонам перед наследованием, когда невозможно определить общий базовый класс; §13.6.1.
21. Отдавайте предпочтение шаблонам перед наследованием, когда использование встроенных типов и структур важно из соображений совместимости; §13.6.1.

13.9. Упражнения

1. (*2) Исправьте ошибки в определении ***List*** из §13.2.5 и напишите C++-код, эквивалентный тому, что сгенерирует компилятор из определения ***List*** и функции ***f***(). Протестируйте работу обоих версий кода. Если вы можете это сделать, то сравните их машинные коды.
2. (*3) Напишите классовый шаблон для односвязного списка с элементами, производными от класса ***Link***, содержащего информацию для связывания элементов. Такой список называется интрузивным. Отталкиваясь от этого списка, напишите код односвязного списка, который может иметь элементы любого типа (неинтрузивный список). Сравните производительность обоих списков и обсудите их достоинства и недостатки.
3. (*2.5) Напишите интрузивный и неинтрузивный двусвязные списки. Какие дополнительные операции нужны этим спискам по сравнению с односвязными вариантами списков?
4. (*2) Завершите шаблон ***String*** из §13.2, основанный на классе ***String*** из §11.12.
5. (*2) Определите шаблонную функцию ***sort***(), принимающую критерий сравнения в качестве шаблонного аргумента. Определите класс ***Record*** с двумя полями — ***count*** и ***price***. Выполните сортировку ***vector***<***Record***> по каждому полю.
6. (*2) Реализуйте шаблон ***qsort***().
7. (*2) Напишите программу, которая читает пары (***key***, ***value***), а затем вычисляет суммы значений ***value***, соответствующих уникальным значениям ***key***. Сформулируйте требования к типам значений ***value*** и ***key***.
8. (*2.5) Реализуйте простой ассоциативный массив Map, основанный на классе ***Assoc*** из §11.8. Убедитесь, что ***Map*** работает корректно как для C-строк, так и для ***string*** в качестве типа ***key***. Убедитесь, что ***Map*** работает корректно как для типов с умолчательными конструкторами, так и для типов без оных. Обеспечьте возможность итерации по элементам контейнера ***Map***.
9. (*3) Сравните производительность программы подсчета слов из §11.8 по сравнению с программой, не использующей ассоциативного массива. В обоих случаях используйте одинаковый ввод/вывод.
10. (*3) Перепрограммируйте ***Map*** из §13.9[8], используя более подходящие структуры данных (например, «красно-черные» деревья).
11. (*2.5) Используйте ***Map*** для выполнения топологической сортировки, описанной в [Knuth,1968] т.1 (второе издание), стр. 262.
12. (*1.5) Обеспечьте корректную работу программы суммирования из §13.9[7] для имен, содержащих пробелы.
13. (*2) Напишите шаблоны ***readline***() для разных форматов строк ввода, например, (item,count,price).

14. (*2) Используйте технику, кратко описанную для ***Literature*** из §13.4 для сортировки строк в обратном лексикографическом порядке. Убедитесь, что указанная техника работает как на системах, где ***char*** является знаковым типом, так и на системах с беззнаковым ***char***. Напишите вариант программы для сортировки без учета регистра символов.
15. (*1.5) Придумайте пример, в котором демонстрируются, по крайней мере, три различия между функциональными шаблонами и макросами (не считая разницы в синтаксисе их определений).
16. (*2) Разработайте схему, гарантирующую, что компилятор проверяет общие ограничения на аргументы шаблонов, для которых создаются объекты. Недостаточно просто проверить ограничения вида «***T*** должно быть типом, производным от ***My_base***».

14

Обработка исключений

Не перебивайте меня,
когда я вас перебиваю.
— Уинстон Черчилль

Обработка ошибок — группировка исключений — перехват исключений — перехват всех исключений — повторная генерация — управление ресурсами — ***auto_ptr*** — исключения и ***new*** — исчерпание ресурсов — исключения в конструкторах — исключения в деструкторах — исключения, не являющиеся ошибками — спецификация исключений — неожиданные исключения — неперехваченные исключения — исключения и эффективность — альтернативные методы обработки ошибок — стандартные исключения — советы — упражнения.

14.1. Обработка ошибок

Как отмечалось в §8.3, автор библиотеки в состоянии обнаружить ошибки времени выполнения, но обычно не имеет представления, что с ними нужно делать. Пользователь библиотеки может знать, как бороться с такими ошибками, но он их не обнаруживает — в противном случае, все было бы сосредоточено в пользовательском коде, а о библиотеке и речь бы не шла. Понятие *исключения* (*exception*) как раз и призвано помочь в разрешении этой коллизии. Центральная идея заключается в том, что функция, обнаружившая ошибку, с которой она не может справиться самостоятельно, *генерирует* (*throws*) исключение в надежде, что вызвавшая ее (непосредственно или косвенно) функция сможет обработать возникшую ошибку. Функции, которые имеют намерение решать проблемы такого рода, должны явно указать, что они *перехватывают* (*catch*) данный тип исключения (§2.4.2, §8.3).

Такой стиль обработки ошибок предпочтительней многих более традиционных способов. Рассмотрим возможные альтернативы. Обнаружив проблему, с которой невозможно справиться локально, функция может:

1. Прекратить выполнение.
2. Возвратить значение, указывающее на ошибку.
3. Вернуть некоторое допустимое значение и оставить программу в ошибочном состоянии.
4. Вызвать функцию, специально предназначенную для вызова в случае ошибки.

Вариант [1], «прекратить выполнение» — это то, что происходит по умолчанию, когда исключения не перехватываются. Для большей части ошибок мы можем и должны придумать что-нибудь получше. В частности, библиотека, которая ничего не знает о цели и общей стратегии использующей ее программы, не может просто так вызвать ***exit***() или ***abort***(). Такую библиотеку, неожиданно завершающую работу всей программы, невозможно использовать в составе надежных приложений. Центральный взгляд на обработку исключений (динамически возникающих ошибочных ситуаций) состоит в том, что нужно управление передавать в вызывающую функцию в тех случаях, когда локально невозможно принять решение об обработке ошибок.

Вариант [2], «возвратить значение, указывающее на ошибку» — не всегда возможен, ибо часто не существует подходящих значений, указывающих на ошибки. Например, для функции с возвратом типа ***int***, любое возвращаемое значение является нормальным, приемлемым результатом. Но даже, если можно выделить специальные «ошибочные возвраты», то все равно такой подход неудобен, поскольку требуется каждый вызов функции проверять на совпадение его возврата с «ошибочными значениями». Это легко может удвоить объем программного кода (§14.8). Как следствие, такой подход редко когда применяют систематически для борьбы со всеми возможными ошибками.

Подход [3], «вернуть некоторое допустимое значение и оставить программу в ошибочном состоянии» — плох тем, что вызывающая функция может не заметить, что программа находится в ненормальном состоянии. Например, многие библиотечные функции языка C для индикации ошибки устанавливают значение глобальной переменной ***errno*** (§20.4.1, §22.3). Однако ж, многие программы не проверяют errno систематически, и в итоге они не застрахованы от последующих ошибок, вызванных использованием значений, полученных от ошибочно отработавших библиотечных функций. Более того, использование глобальных переменных для индикации ошибок проблематично при работе нескольких программ в параллельных процессах.

Обработка исключений не тождественна случаям, когда уместен вариант [4] — «вызвать функцию, специально предназначенную для вызова в случае ошибки». В отсутствие обработки исключений, вариант [4] имеет те же самые три альтернативы в плане того, как именно специальная функция для обработки ошибок выполняет эту работу. Дальнейшее обсуждение функций обработки ошибок и исключений см. в §14.4.5.

Механизм обработка исключений предоставляет альтернативу традиционным методам в случаях, когда последние не достаточны, не элегантны или сами подвержены ошибкам. Он формулирует способ явного отделения кода обработки ошибок от регулярного кода программы, что способствует лучшей читаемости программы человеком и большей удобоваримости ее кода для вспомогательных инструментальных средств. Механизм обработки исключений предоставляет более регуляр-

ную технологию обработки ошибок, упрощающую взаимодействие между раздельно написанными программными модулями.

Один аспект, принятый в схеме обработки исключений, и состоящий в том, что умолчательной реакцией на ошибки (особенно на ошибки в библиотеках) является завершение работы программы, необычен для программистов на языках С или Pascal. Для них традиционной реакцией является попытка «тянуть программу далее, надеясь на лучшее». Этот аспект механизма обработки исключений делает использующие его программы более капризными в том смысле, что он требует больших усилий и внимания для того, чтобы довести работу программы до конца. Однако такой подход более предпочтителен, ибо значительно лучше бороться с проблемами на ранних этапах разработки, чем на поздних, и уж тем более нежелательно передавать якобы готовую программу ничего не подозревающему конечному пользователю, чтобы именно он столкнулся с этими проблемами. Когда завершение работы программы неприемлемо, можно осуществлять перехват всех исключений (§14.3.2) или некоторой их точно специфицированной части (§14.6.2). Таким образом, исключения прекращают выполнение программ только тогда, когда программист позволяет им это делать. Это значительно лучше, чем безоговорочное завершение программ в случаях, когда недостаточная обработка ошибок приводит к их катастрофическому состоянию.

Иногда программисты пытаются смягчить последствия подхода «тянуть до конца» путем вывода сообщений об ошибках, отображением диалоговых окон, обращением к пользователю за подмогой и т.д. Такие подходы действительно полезны при отладке программ, когда пользователем является сам программист, знакомый со структурой программы. Для обычного же пользователя библиотека, запрашивающая помощь у (возможно отсутствующего) пользователя/оператора, неприемлема. Кроме того, в некоторых случаях сообщениям об ошибке просто негде появиться (скажем, если программа исполняется в системах, где поток cerr не подключен к средствам визуализации), но и без этого конечному пользователю они не понятны. Далее, сообщения могут быть на естественном языке, непонятном пользователю (скажем, на финском языке — для пользователя-англичанина). Что еще хуже, сообщения об ошибках от библиотек могут касаться ее внутренних реалий, незнакомых пользователю (например, сообщение «недопустимый аргумент для ***atan2***» в ответ на неправильный ввод для графической системы). Хорошие библиотеки не должны так «разговаривать с пользователями». Исключения позволяют коду, обнаружившему проблему, но не знающему, что с ней делать, отпасовать ее в другую часть системы, которая способна принять правильное решение. И только эта часть системы, которая видит весь контекст исполнения программы, может также и сформировать осмысленное сообщение об ошибке.

Механизм обработки исключений можно рассматривать как аналог механизма выявления ошибок типов и разрешения неоднозначностей на стадии компиляции, только механизм исключений работает на стадии выполнения программы. Обработка исключений делает более важным этап проектирования и может увеличить объем работ, необходимых для получения начальных, работающих с ошибками версий программы. Однако в результате получается конечный код, у которого намного больше шансов работать как задумано, как часть большой программы, понятной другим программистам, и который доступен для автоматической обработки различными инструментальными средствами. Как и другие средства языка С++, обработ-

ка исключений предоставляет программисту средства для хорошего стиля программирования, который можно лишь неформально и не в полном объеме практиковать в таких языках как C или Pascal.

Надо понимать, что обработка ошибок является сложной задачей и что механизм обработки исключений, даже с учетом его большей формализации по сравнению с альтернативными методами, все же менее структурирован на фоне иных средств языка, имеющих дело лишь с локальным контекстом потока выполнения программы. Механизм обработки исключений предоставляет программисту возможность обрабатывать ошибки там, где их обрабатывать наиболее сподручно для конкретной системы. Исключения позволяют явным образом продемонстрировать всю сложность обработки ошибок. Но не они являются первопричиной этой сложности (не следует обвинять гонца с плохими новостями).

Сейчас самое подходящее время для повторного прочтения §8.3, где изложены базовый синтаксис, семантика и приемы использования исключений.

14.1.1. Альтернативный взгляд на исключения

Исключения — это такой термин, который разные люди понимают по-разному. Механизм исключений языка C++ разработан для обработки ошибок и других исключительных ситуаций (отсюда название механизма). В частности, он поддерживает обработку ошибок в программах, составленных из независимо разработанных программных компонентов.

Механизм исключений предназначен для обработки лишь синхронных исключений, таких как выход за границы допустимых диапазонов в массивах или ошибки ввода/вывода. Асинхронные события, такие как прерывания от клавиатуры и ряд арифметических ошибок, не обязательно являются исключениями и не обрабатываются непосредственно обсуждаемым механизмом. Асинхронные события требуют для своей обработки механизмов, принципиально отличающихся от механизма исключений (как мы его здесь определили). На многих системах предлагаются механизмы работы с асинхронными событиями, например сигналы, но поскольку эти механизмы системно-зависимые, то они здесь не рассматриваются.

Механизм обработки исключений представляет собой нелокальную управляющую структуру, базирующуюся на *раскрутке стека* (*stack unwinding*) (§14.4), которую можно рассматривать как альтернативный возврат из функции. Вполне законно использовать исключения, которые не имеют никакого отношения к обработке ошибок (§14.5). Но, конечно же, первичной целью этого механизма и предметом данной главы является именно обработка ошибок и обеспечение надежности программных систем.

В стандарте C++ нет понятий потока и процесса. Следовательно, обстоятельства функционирования механизма исключений применительно к потокам и процессам здесь не рассматриваются. Средства параллельного выполнения, доступные на вашей системе, описываются в ее документации. Здесь я просто отмечаю, что механизм исключений разработан таким образом, чтобы быть эффективным и в среде параллельного исполнения программ при соблюдении программистом таких правил, как блокирование разделяемых структур данных при доступе к этим данным.

Механизм обработки исключений в C++ предназначен для того, чтобы обрабатывать ошибки и исключительные ситуации, и информировать о них. Задача программиста состоит в том, чтобы решить, какие ситуации являются исключительными для данной конкретной программы. Это не так легко сделать (§14.5). Можно ли считать исключительным событие, которое происходит почти что в каждом сеансе работы программы? Может ли запланированное и обрабатываемое событие считаться ошибкой? Ответ на оба вопроса — да. «Исключительное» не значит редко встречающееся или катастрофическое. Лучше рассматривать исключительные ситуации как «невозможность некоторой части системы выполнить то, что от нее требуется». Обычно в таких случаях следует предпринять что-то иное. Генерация исключений с помощью ***throw*** должна быть более редкой по сравнению с обычными вызовами функций, иначе структура программы станет неотчетливой. В то же время, мы можем твердо рассчитывать на то, что большинство грамотно спроектированных крупных программ наверняка генерируют и перехватывают некоторое количество исключений по ходу своего нормального и вполне успешного выполнения.

14.2. Группировка исключений

Исключение является объектом некоторого класса, представляющим исключительную ситуацию. Обнаруживший ошибку код (часто это библиотека) генерирует исключение оператором ***throw*** (§8.3). Часть кода, осуществляющая обработку исключений, сосредотачивается в блоке ***catch***. Оператор ***throw*** порождает раскрутку стека до тех пор, пока не будет найден подходящий ***catch-блок*** (в функции, прямо или косвенно вызвавшей функцию с этим оператором ***throw***).

Часто исключения разбиваются на семейства. Это означает, что наследование может отражать такое структурирование исключений и помогать в их обработке. Например, исключения в математической библиотеке могут быть организованы следующим образом:

```
class Matherr { };
class Overflow : public Matherr { };
class Underflow : public Matherr { };
class Zerodivide : public Matherr { };
// ...
```

Это позволяет нам обрабатывать любое исключение как ***Matherr***, не заботясь о точном его типе. Например:

```
void f()
{
  try
  {
    // ...
  }
  catch (Overflow)
  {
    // обработка Overflow или производных от Overflow
  }
```

```
  catch (Matherr)
  {
    // обработка любых Matherr кроме Overflow
  }
}
```

Здесь исключения типа ***Overflow*** обрабатываются специальным образом. А все остальные исключения из представленной иерархии подходят для обобщенной (одинаковой) обработки.

Организация исключений в виде наследственной иерархии может быть полезна для обеспечения надежности кода. Например, как бы вам пришлось обрабатывать исключения математической библиотеки без представленной выше иерархии? Ясно, что потребовалось бы занудное перечисление всех типов исключений:

```
void g ()
{
  try
  {
    //...
  }
  catch (Overflow) { /* ... */ }
  catch (Underflow) { /* ... */ }
  catch (Zerodivide) { /* ... */ }
}
```

И это не только утомительно. Программист вообще может при этом забыть какое-либо исключение. Далее, если мы добавим новое исключение в математическую библиотеку, каждый фрагмент клиентского кода, обрабатывающий все математические исключения, потребовал бы модификации. Как правило, такие глобальные модификации недопустимы после выхода первой рабочей версии библиотеки. Часто они вообще невозможны, если не удается найти все необходимые куски кода. Рассмотренные проблемы сопровождения кода фактически ставят крест на попытках модификации системы исключений в библиотеках после их первоначального опубликования, что неприемлемо для большинства библиотек. Из рассмотренного вытекает, что в библиотеках или изолированных частях программных систем исключения должны организовываться в наследственные иерархии (§14.6.2).

Обратите внимание на то, что ни встроенные математические операции, ни базовые библиотечные математические функции (языков C/C++) не обрабатывают арифметические ошибки как исключения. Одной из причин такого явления служит факт, что арифметические ошибки вроде деления на нуль проявляются асинхронно на многих машинах с конвейерной архитектурой.

Приведенная выше иерархия исключений ***Matherr*** является просто иллюстрацией. Стандартные библиотечные исключения языка C++ описываются в §14.10.

14.2.1. Производные исключения

Использование иерархий классов для обработки исключений естественным образом приводит к обработчикам, интересующимся лишь подмножеством информации, которую несут с собой исключения. Другими словами, в типичном случае ис-

ключение перехватывается обработчиком его базового класса, а не обработчиком его собственного класса. Семантика именования и перехвата исключений аналогична таковой для функций с аргументом. То есть формальный аргумент инициализируется фактическим значением вызова (§7.2). Это означает, что сгенерированное исключение «срезается» (sliced) до перехваченного (§12.2.3). Например:

```
class Matherr
{
  // ...
  virtual void debug_print() const { cerr << "Math error" ; }
};

class Int_overflow: public Matherr
{
  const char* op;
  int a1, a2;

public:
  Int_overflow (const char* p, int a, int b) { op = p; a1 = a; a2 = b; }
  virtual void debug_print() const { cerr << op << ' (' << a1 << ', ' << a2 << ') '; }
  // ...
};

void f()
{
  try
  {
    g();
  }
  catch (Matherr m)
  {
    // ...
  }
}
```

Когда вызывается обработчик для ***Matherr***, ***m*** является объектом типа ***Matherr*** — даже если функция ***g***() во время своей работы сгенерировала исключение типа ***Int_overflow***. Это означает, что добавочная информация, сопутствующая типу ***Int_overflow***, недоступна.

Как всегда, можно использовать указатели или ссылки во избежание потери информации. Например:

```
int add(int x, int y)
{
  if( (x>0 && y>0 && x>INT_MAX-y) || (x<0 && y<0 && x<INT_MIN-y) )
    throw Int_overflow("+",x,y);

  return x+y;                     // вычисление x+y не вызовет исключений
}

void f()
{
  try
  {
```

```
    int i1 = ad(1,2);
    int i2 = add(INT_MAX,-2);
    int i3 = add(INT_MAX,2);   // приехали!
  }
  catch (Matherr& m)
  {
    // ...
    m.debug_print();
  }
}
```

В результате последнего из указанных вызовов ***add***() генерируется исключение, приводящее к вызову метода ***Int_overflow::debug_print***(). Если бы исключение перехватывалось «по значению», а не «по ссылке», то был бы вызван метод ***Matherr::debug_print***().

14.2.2. Композитные (комбинированные) исключения

Не каждая группа исключений обязана образовывать древовидную структуру. Бывает, что исключение одновременно принадлежит сразу двум группам. Например:

```
class Netfile_err : public Network_err, public File_system_err { /* ... */ };
```

Исключения ***Netfile_err*** могут перехватываться как функциями, настроенными на работу с сетевыми исключениями

```
void f()
{
  try
  {
    // что-либо
  }
  catch (Network_err& e)
  {
    // ...
  }
}
```

так и функциями, настроенными на работу с файловыми исключениями:

```
void g()
{
  try
  {
    //...
  }
  catch (File_system_err& e)
  {
    // ...
  }
}
```

Такие комбинированные (композитные) исключения важны в тех случаях, когда, например, сетевые службы прозрачны для пользователей — автор функции ***g***()

в нашем примере мог не догадываться о том, что сетевые службы вовлечены в дело (см. также §14.6).

14.3. Перехват исключений

Рассмотрим пример:

```
void f()
{
  try
  {
    throw E() ;
  }
  catch (H)
  {
    // когда мы сюда попадем?
  }
}
```

Обработчик исключений будет вызван в следующих случаях:

1. Если ***H*** того же типа, что и ***E***.
2. Если ***H*** — это однозначный базовый класс для ***E***.
3. Если ***H*** и ***E*** — указательные типы, и [1] или [2] справедливы для типов, на которые указывают эти указатели.
4. Если ***H*** — это ссылка, и [1] или [2] справедливы для типа, на который ***H*** ссылается.

Мы также можем добавить модификатор ***const*** к типу исключения, указанному в блоке ***catch*** (точно так же, как это делается в отношении аргументов функций). Это не изменит набор исключений, который мы можем перехватить, а лишь оградит нас от изменения объекта исключения.

В принципе, исключение при генерации копируется, так что обработчики имеют дело с копиями оригинальных исключений. Исключение может быть даже несколько раз скопировано перед тем, как будет перехвачено для обработки. Отсюда следует, что мы не можем сгенерировать некопируемое исключение. Конкретные реализации могут применять разные стратегии для хранения и передачи исключений. Всегда, однако, гарантируется, что хватит памяти для генерации операцией new стандартного исключения нехватки памяти ***bad_alloc*** (§14.4.5).

14.3.1. Повторная генерация исключений

Перехватив исключение, обработчик рассматривает его и может иногда прийти к выводу, что сам он не в состоянии выполнить исчерпывающую обработку этого исключения. Тогда он делает ту часть работы, которая ему по плечу, после чего вновь генерирует это же исключение. В результате, исключение может обрабатываться там, где это наиболее уместно. Это возможно даже в случаях, когда необходимая для обработки исключения информация не сосредоточена в единственном месте программы, и лучше всего обработку рассредоточить по нескольким обработчикам. Например:

```
void h ()
{
  try
  {
    // код, который может генерировать "математические" исключения
  }
  catch (Matherr)
  {
    if(can_handle_it_completely)
    {
      // обработать Matherr
      return;
    }
    else
    {
      // делаем здесь то, что можем
      throw;      // повторно генерируем исключение
    }
  }
}
```

Повторная генерация исключения обозначается ключевым словом ***throw***, за которым не указывается никакого выражения. Если пытаться выполнить такой оператор в отсутствие исключений, то будет вызвана функция ***terminate***() (§14.7). Компиляторы могут (но не всегда) предупреждать о таких ошибках.

Повторно сгенерированное исключение есть точная копия оригинального исключения, а не его часть, что была доступна как ***Matherr***. Другими словами, если было сгенерировано исключение типа ***Int_overflow***, то функция, вызвавшая ***h***(), может по-прежнему перехватывать исключение типа ***Int_overflow***, несмотря на то, что в самой функции ***h***() исключение перехвачено с типом ***Matherr***.

14.3.2. Перехват любых исключений

Бывает полезна и вырожденная версия описанной выше техники повторной генерации исключений. Как и для функций, где многоточие означает «любой аргумент» (§7.6), ***catch***(...) означает «перехватить любое исключение». Например:

```
void m ()
{
  try
  {
    // что-либо
  }
  catch (...)   // обработка любых исключений
  {
    // обработка (очистка)
    throw;
  }
}
```

То есть если основная часть функции ***m***() сгенерирует любое исключение, в обработчике будут выполнены необходимые при этом завершающие действия (очист-

ка — cleanup). После выполнения локальной очистки исключение генерируется заново, чтобы имелась возможность выполнить в иных местах программы дополнительную обработку исходной ошибки. В §14.6.3.2 рассмотрена техника получения информации об исключении в обработчике с многоточием.

Важным аспектом обработки ошибок вообще и обработки исключений в частности является поддержка инвариантов программы (§24.3.7.1). Например, если предполагается, что *m*() оставляет некоторые указатели в том же состоянии, в котором она их получила, то в обработчике можно написать код, присваивающий им надлежащие значения. Таким образом, обработчик любых типов исключений может использоваться для поддержки произвольных инвариантов. Тем не менее, такое решение не является для многих важных случаев самым элегантным (§14.4).

14.3.2.1. Порядок записи обработчиков

Поскольку исключение производного типа может перехватываться несколькими обработчиками, порядок, в котором в коде представлены эти обработчики, важен. Дело в том, что отбор обработчиков выполняется в соответствие с этим порядком. Например:

```
void f()
{
  try
  {
    // ...
  }
  catch(std::ios_base::failure)
  {
    // обработка любых ошибок потокового ввода/вывода (§14.10)
  }
  catch(std::exception& e)
  {
    // обработка любых исключений стандартной библиотеки (§14.10)
  }
  catch(...)
  {
    // обработка любых других исключений (§14.3.2)
  }
}
```

Поскольку компилятор разбирается в иерархии классов, он может выявить многие логические ошибки. Например:

```
void g()
{
  try
  {
    // ...
  }
  catch(...)
  {
    // обработка любых исключений (§14.3.2)
  }
```

```
    catch (std::exception& e)
    {
        // обработка любых исключений стандартной библиотеки (§14.10)
    }
    catch (std::bad_cast)
    {
        // обработка исключения из операции dynamic_cast (§15.4.2)
    }
}
```

Здесь обработчик исключений типа ***exception*** никогда не будет выполняться. Даже если мы уберем обработчик с многоточием, то ***bad_cast*** никогда не будет рассматриваться, потому что тип ***bad_cast*** является производным от ***exception***.

14.4. Управление ресурсами

Если функция получает некоторый ресурс (открывает файл, динамически получает блок свободной памяти, устанавливает блокировку доступа и т.д.), то для будущего функционирования системы очень важно, чтобы этот ресурс был впоследствии корректно освобожден. Часто сама функция и освобождает ресурсы перед возвратом. Например:

```
void use_file (const char* fn)
{
    FILE* f=fopen (fn, "r") ;
    // используем f
    fclose (f) ;
}
```

Это выглядит приемлемым решением до того момента, как мы задумаемся о том, что будет в случае возникновения исключения после вызова ***fopen***() и до вызова ***fclose***()? Ведь в этом случае возможен выход из ***use_file***() без вызова ***fclose***(). Эта проблема характерна для языков программирования, не содержащих встроенных средств обработки исключений. Например, стандартная библиотечная функция ***longjmp***() языка C запросто может вызвать подобного рода проблемы. Даже простейший оператор return может завершить ***use_file***(), оставив файл открытым.

Первая попытка сделать функцию ***use_file***() устойчивой к таким ошибкам выглядит следующим образом:

```
void use_file (const char* fn)
{
    FILE* f=fopen (fn, "r") ;

    try
    {
        // используем f
    }
    catch (...)
    {
        fclose (f) ;
```

```
    throw;
  }
  fclose(f);
}
```

Использующий файл код заключен в блок *try*, что обеспечивает здесь перехват любых исключений, закрытие файла и повторную генерацию исключения.

Проблема с этим решением состоит в том, что оно слишком многословное, утомительное и потенциально дорогостоящее. Более того, любое многословное и утомительное решение подвержено ошибкам, ибо программисту оно надоедает. Но к счастью, существует более элегантное решение. Представим общую постановку проблемы в следующем виде:

```
void acquire()
{
  // выделяем ресурс 1
  // ...
  // выделяем ресурс n
  // используем ресурсы
  // освобождаем ресурс n
  // ...
  // освобождаем ресурс 1
}
```

В типичном случае ресурсы должны освобождаться в порядке, обратном порядку их выделения. Это сильно напоминает поведение локальных объектов, создаваемых конструкторами и уничтожаемых деструкторами. Следовательно, проблему получения и освобождения ресурсов можно решить с помощью объектов классов, располагающих надлежащими конструкторами и деструкторами. Например, мы могли бы определить класс *File_ptr*, ведущий себя как указатель на файл (то есть как тип *FILE**):

```
class File_ptr
{
  FILE* p;

public:
  File_ptr(const char* n, const char* a) {p = fopen(n,a);}
  File_ptr(FILE* pp) {p = pp;}
  // подходящие операции копирования
  ~File_ptr() { if(p) fclose(p); }

  operator FILE*() {return p;}
};
```

Объекты типа *File_ptr* создаются либо на базе *FILE**, либо на базе информации, необходимой функции *fopen()*. В любом случае, объект будет автоматически уничтожен по выходу из области видимости, а деструктор закроет ассоциированный с ним файл. Наша функция сократилась теперь до минимума:

```
void use_file(const char* fn)
{
  File_ptr f(fn,"r");
  // используем f
}
```

Деструктор будет вызван независимо от того, завершается ли функция нормальным образом или по исключению. Таким образом, механизм обработки исключений позволил нам удалить код обработки ошибок из реализации основного алгоритма задачи. Результирующий код проще и менее подвержен внесению случайных ошибок, чем традиционный вариант.

Процесс восходящей обработки стека вызовов с целью обнаружения подходящего обработчика исключения обычно называют *раскруткой стека* (*stack unwinding*). По мере постепенной раскрутки стека вызовов для всех локально сконструированных в них объектов гарантированно вызываются деструкторы.

14.4.1. Использование конструкторов и деструкторов

Техника управления ресурсами через локальные объекты называется так: «получение ресурса есть инициализация». Эта весьма общая техника опирается на свойства конструкторов и деструкторов в связи с их взаимодействием с исключениями.

Объект не считается полностью созданным до тех пор, пока не завершится работа конструктора над этим объектом. Только после этого механизм раскрутки стека с вызовом деструктора станет возможным. Объект, состоящий из подобъектов, считается полностью сконструированным лишь тогда, когда сконструированы все его подобъекты. Создание массива включает создание всех его элементов (и лишь полностью сконструированные элементы массива корректно уничтожаются в процессе раскрутки стека).

Конструктор выполняет попытку корректно и в полном объеме сконструировать объект. Когда же не удается это сделать, хорошо написанный конструктор восстанавливает (насколько возможно) исходное состояние системы. В идеале, конструкторы должны реализовывать одну из этих альтернатив, никогда не оставляя объекты в полуподготовленном состоянии. Этого можно добиться, применяя к членам класса принцип «получение ресурса есть инициализация».

Рассмотрим класс *X*, конструктор которого должен получить два ресурса — файл *x* и блокировку *y*. В процессе получения ресурсов может генерироваться исключение, сигнализирующее о недоступности ресурса. Конструктор класса *X* не должен нормальным образом завершать работу в том случае, когда файл получен, а блокировка оказалась недоступной. Эту проблему можно решить элегантно (без излишней нагрузки на программиста). Для приобретения указанных ресурсов, используем классы ***File_ptr*** и ***Lock_ptr***, соответственно. Получение ресурса формулируем в виде инициализации локальных объектов, представляющих эти ресурсы:

```
class X
{
  File_ptr aa;
  Lock_ptr bb;

public:
  X(const char* x, const char* y)
      : aa(x,"rw"),          // приобретаем 'x'
      : bb(y)                // приобретаем 'y'
  {}
  //...
};
```

Теперь, как и для локальных объектов, реализация C++ берет на себя все заботы о деталях выполнения процесса (никаких дополнительных действий от программиста не требуется). Например, если исключение возникнет после того, как прошла инициализация ***aa***, но до того, как создан ***bb***, будет автоматически вызван деструктор для ***aa*** (но не для ***bb***).

Итак, где годится рассмотренная простая модель получения ресурсов, от программиста не требуется писать явный код обработки исключений.

Самым распространенным ресурсом является память. Например:

```
class Y
{
  int* p;
  void init() ;

public:
  Y(int s)  { p = new int[s] ; init() ; }
  ~Y() {delete [] p; }
  // ...
};
```

Такая практика весьма распространена и она может приводить к «утечкам памяти» (memory leaks). Если в ***init***() будет сгенерировано исключение, то выделенная в конструкторе память не освобождается — деструктор не будет вызван, ибо объект сконструирован к этому моменту не полностью. Вот более безопасный вариант:

```
class Z
{
  vector<int> p;
  void init() ;

public:
  Z(int s)  : p(s)  { init() ; }
  // ...
};
```

Память, используемая вектором ***p***, теперь находится под управлением типа ***vector***. Если ***init***() сгенерирует исключение, выделенная память будет (неявно) освобождена вызовом деструктора для ***p***.

14.4.2. Auto_ptr

Стандартная библиотека предоставляет классовый шаблон ***auto_ptr***, поддерживающий технику «получение ресурса есть инициализация». В своей основе ***auto_ptr*** инициализируется указателем и может быть разыменован аналогично указателю. Кроме того, указуемый объект неявно удаляется (уничтожается) тогда, когда объект типа ***auto_ptr*** выходит из области видимости. Например:

```
void f(Point p1, Point p2, auto_ptr<Circle> pc, Shape* pb)
{
  auto_ptr<Shape> p(new Rectangle(p1,p2)) ;  // p указывает на rectangle
  auto_ptr<Shape> pbox(pb) ;

  p->rotate(45) ;                              // используем auto_ptr<Shape> как Shape*
  // ...
```

```
    if(in_a_mess) throw Mess() ;
    //...
}
```

Здесь объекты типа ***Rectangle***, ***Shape*** (адресуется через ***pb***) и ***Circle*** (адресуется с помощью ***pc***) корректно уничтожаются независимо от того, генерируется исключение или же не генерируется.

Для достижения *семантики владения* (*ownership semantics*), копирование объектов типа ***auto_ptr*** радикально отличается от копирования обычных указателей (реализуется *семантика деструктивного копирования — destructive copy semantics*): если некоторый объект типа ***auto_ptr*** копируется в другой объект того же типа, то исходный объект после этого уже ни на что не указывает. Из-за того, что копирование объектов типа ***auto_ptr*** изменяет их самих, нельзя скопировать объекты типа ***const auto_ptr***.

Шаблон ***auto_ptr*** определен в файле <***memory***>:

```
template <class Y> struct auto_ptr_ref { /*...*/ };        // вспомогательный класс

template<class X> class std::auto_ptr
{
  X* ptr;
public:
  typedef X element_type;

  // внимание: копирующий конструктор и присваивания имеют не-const аргументы:
  explicit auto_ptr(X* p=0) throw() {ptr=p;}   // throw()-"ничего не генерировать"; см.§14.6
  auto_ptr (auto_ptr& a) throw () ;                        // копируем, затем a.ptr=0
  template<class Y> auto_ptr(auto_ptr<Y>& a) throw() ;     // копируем, затем a.ptr=0
  ~auto_ptr( ) throw() { delete ptr; }

  auto_ptr& operator=(auto_ptr& a) throw() ;                           // копируем и a.ptr=0
  template<class Y> auto_ptr& operator=(auto_ptr<Y>& a) throw() ;      // копируем и a.ptr=0
  template<class Y> auto_ptr& operator=(auto_ptr_ref<Y>& a) throw() ;  // копируем и release

  X& operator*() const throw() { return *ptr; }
  X* operator->() const throw() { return ptr; }

  X* get() const throw() { return ptr; }                      // извлечь указатель
  X* release() throw() { X* t=ptr; ptr=0; return t; }         // передача владения
  void reset(X* p =0) throw() {if(p!=ptr) {delete ptr; ptr=p; } }

  auto_ptr(auto_ptr_ref<X>) throw () ;                        // копирование из auto_ptr_ref
  template<class Y> operator auto_ptr_ref<Y>() throw() ;      // копирование в auto_ptr_ref
  template<class Y> operator auto_ptr<Y>() throw () ;         // деструктив. коп-е из auto_ptr
};
```

Цель ***auto_ptr_ref*** — реализовать семантику деструктивного копирования для ***auto_ptr***, делая невозможным копирование ***const auto_ptr***. Если ***D**** может быть преобразован в ***B****, тогда конструктор и операция присваивания шаблона могут (явно или неявно) преобразовывать ***auto_ptr<D>*** в ***auto_ptr<B>***. Например:

```
void g(Circle* pc)
{
  auto_ptr<Circle> p2(pc) ;  // теперь p2 отвечает за удаление
  auto_ptr<Circle> p3(p2) ;  // теперь p3 отвечает за удаление (а p2 нет)
```

```
    p2->m = 7;                  // ошибка программиста: p2.get()==0
    Shape* ps = p3.get();       // извлекаем указатель из auto_ptr

    auto_ptr<Shape> aps(p3);// передаем владение и преобразуем тип
    auto_ptr<Circle> p4(pc); // ошибка программиста: теперь и p4 отвечает за удаление
}
```

Эффект принадлежности объекта двум ***auto_ptr*** не определен; скорее всего, это приведет к двукратному уничтожению объекта (с отрицательными последствиями).

Отметим, что из-за семантики деструктивного копирования объекты типа ***auto_ptr*** не удовлетворяют требованиям для элементов стандартных контейнеров и стандартных алгоритмов, таких как ***sort***(). Например:

```
vector<auto_ptr<Shape> >& v;  //опасно: использование auto_ptr в контейнере
// ...
sort(v.begin(),v.end());       //не делайте этого: сортировка может испортить v
```

Ясно, что ***auto_ptr*** не является интеллектуальным указателем (smart pointer); он лишь без дополнительных накладных расходов реализует сервис, для которого и был задуман — повышенную надежность (безопасность) кода в связи с возможностью исключений.

14.4.3. Предостережение

Не все программы должны обладать повышенной живучестью по отношению к любым видам ошибочных ситуаций, и не все ресурсы столь критичны, чтобы оправдать совокупные усилия для применения принципа «получение ресурса есть инициализация», типа ***auto_ptr*** и механизма ***catch***(...). Например, для большинства программ, которые просто читают ввод и далее работают в направлении своего завершения, наиболее естественной реакцией на возникновение серьезной ошибки будет прекращение работы (после выдачи приемлемого диагностического сообщения). Это позволяет системе освободить ранее выделенные ресурсы, а пользователю — запустить программу заново и ввести корректные данные. Рассмотренная же стратегия предназначена для приложений, которые не могут просто так завершить работу. В частности, разработчик библиотеки не может знать требований программы, использующей библиотеку, в отношении стратегии поведения в случае возникновения ошибочных ситуаций. В результате, он вынужден избегать любых форм безусловного прекращения работы и вынужден заботиться об освобождении ранее выделенных ресурсов перед возвратом в головную программу. Для таких вот случаев и подходит стратегия «получение ресурса есть инициализация» в совокупности с исключениями для сигнализации об ошибочном ходе выполнения библиотечных функций.

14.4.4. Исключения и операция new

Рассмотрим пример:

```
void f(Arena& a, X* buffer)
{
  X* p1 = new X;
  X* p2 = new X[10];
```

```
    X* p3 = new (&buffer[10]) X;   // поместить X в buffer (освобождение не нужно)
    X* p4 = new (&buffer[11]) X[10];

    X* p5 = new (a) X;             // выделить из Arena a  (освободить из Arena a)
    X* p6 = new (a) X[10];
}
```

Что произойдет, если конструктор класса *X* сгенерирует исключение? Освобождается ли память, выделенная функцией ***operator new*** ()? В обычном случае ответ положителен, так что инициализации ***p1*** и ***p2*** не вызывают утечки памяти.

Когда же используется синтаксис размещения (§10.4.11), ответ не так прост; некоторые варианты применения этого синтаксиса включают выделение памяти, подлежащей освобождению, а другие не включают. Более того, главной целью синтаксиса размещения является возможность нестандартного выделения памяти, так что и ее освобождение должно быть нестандартным. Следовательно, необходимые действия зависят от используемой схемы выделения памяти: если для выделения памяти вызывалась функция ***Z*::*operator new***(), то требуется вызвать функцию ***Z*::*operator delete***() (если она определена), а в противном случае память освобождать не нужно. Массивы обрабатываются аналогичным образом (§15.6.1). Данная стратегия работает как с операцией «размещающее ***new***» из стандартной библиотеки (§10.4.11), так и в случае, когда пользователь сам реализовал функции для выделения/освобождения памяти.

14.4.5. Исчерпание ресурсов

То и дело возникает вопрос, что делать, когда попытка получить ресурс завершилась неудачно? Например, мы можем с легкой душой открывать файл (используя функцию ***fopen***()) или запрашивать блок свободной памяти (с помощью операции ***new***), даже не побеспокоившись о том, что будет, если файла нет в указанном месте или если нет достаточного объема свободной памяти. Сталкиваясь с такими проблемами, программист может выбрать одну из двух стратегий реагирования:

- *Возобновление* (*resumption*): попросить вызвавшую функцию справиться с проблемой и продолжить работу.
- *Завершение* (*termination*): бросить подзадачу и вернуть управление вызвавшей функции.

В рамках первой стратегии вызвавшая функция должна быть готова помочь решить проблему с выделением ресурса (неизвестным участком кода), а во втором случае — она должна быть готова отказаться от затребованного ресурса. Второй вариант в большинстве случаев намного проще и позволяет лучше реализовать разделение уровней абстракции. Обратите внимание, что при этом речь идет не о прекращении работы программы, а лишь о прекращении частной подзадачи. Стратегия завершения означает стратегию возврата из места, где некоторые вычисления окончились неудачей, в обработчик ошибок, связанный с вызвавшей функцией (может, например, повторно запустить неудавшиеся ранее вычисления), а не попытку устранения проблемы и возобновление вычислений с места, где проблема была обнаружена.

В языке C++ стратегия возобновления поддерживается механизмом вызова функций, а стратегия завершения — механизмом обработки исключений. Обе стра-

тегии можно проиллюстрировать простым примером реализации и использования функции ***operator new***():

```
void* operator new (size_t size)
{
 for (;;)
  {
   if(void* p = malloc (size) ) return p;          // пытаемся найти память
   if(_new_handler == 0) throw bad_alloc () ;    // нет обработчика: сдаемся
   _new_handler () ;                              // запрашиваем помощь
  }
}
```

Здесь я использую функцию ***malloc***() из стандартной библиотеки языка C для выполнения поиска свободной памяти; другие реализации функции ***operator new***() могут избрать иные способы для выполнения такой работы. Если память выделена, ***operator new***() может возвратить указатель на нее. В противном случае вызывается функция-обработчик ***_new_handler***(), призванная помочь отыскать дополнительный объем свободной памяти. Если она в этом преуспевает, то все прекрасно. Если нет — то обработчик не может просто вернуть управление, так как это приведет к бесконечному циклу. Он должен сгенерировать исключение, предоставив вызывающей программе разбираться с проблемой:

```
void my_new_handler ()
{
 int no_of_bytes_found = find_some_memory () ;
 if(no_of_bytes_found < min_allocation) throw bad_alloc () ;     // сдаемся
}
```

Где-то должен быть соответствующий ***try***-блок:

```
try
{
 // ...
}
catch (bad_alloc)
{
 // как-нибудь реагируем на исчерпание памяти
}
```

В функции ***operator new***() ***_new_handler*** является указателем на функцию-обработчик, регистрируемую с помощью стандартной функции ***set_new_handler***(). Если мне нужно использовать для этой цели функцию ***my_new_handler***(), я могу написать:

```
set_new_handler (&my_new_handler) ;
```

Если же еще и нужно перехватывать исключение ***bad_alloc***, то тогда следует написать такой код:

```
void f()
{
 void (*oldnh) () = set_new_handler (&my_new_handler) ;
```

```
    try
    {
      // ...
    }
    catch (bad_alloc)
    {
      // ...
    }
    catch ( . . . )
    {
      set_new_handler (oldnh) ;    // вернуть обработчик
      throw;                       // повторно сгенерировать исключение
    }

    Set_new_handler (oldnh) ;      // вернуть обработчик
}
```

А еще лучше вместо ***catch*** (. . .) использовать стратегию «получение ресурса есть инициализация» (см. §14.4) применительно к ***_new_handler*** (§14.12[1]).

При использовании ***_new_handler*** никакой дополнительной информации не передается из места обнаружения ошибки в функцию-обработчик. В принципе, передать дополнительную информацию не сложно. Но чем больше дополнительной информации при этом передается, тем больше взаимозависимость этих двух фрагментов кода. Это означает, что изменения в одном фрагменте требуют детального знания другого фрагмента, а может быть даже и изменения второго фрагмента. Поэтому для достижения минимума зависимости между фрагментами лучше ограничивать объем пересылаемой между ними информации. Механизм обработки исключений обеспечивает лучшую изоляцию фрагментов, чем механизм вызова функций-обработчиков, предоставляемых вызывающей стороной.

Как правило, лучше организовать выделение ресурсов послойно (в разных уровнях абстракции), минимизируя зависимости одних слоев от помощи со стороны других (вызывающих) слоев. Опыт работы с крупными системами показывает, что они эволюционируют в указанном направлении.

Генерации исключений сопутствует создание объектов. Конкретные реализации в ситуации исчерпания памяти обязаны гарантировать создание объекта типа ***bad_alloc***. Однако генерация иных исключений может приводить к полному исчерпанию памяти.

14.4.6. Исключения в конструкторах

Исключения решают проблему индикации ошибок в работе конструкторов. Ввиду того, что конструкторы не возвращают никаких значений, традиционными альтернативами исключений являются следующие подходы:

1. Оставить создаваемый объект в нештатном состоянии, в надежде на проверку его состояния пользователем.
2. Установить значение нелокальной переменной (например, ***errno***) для индикации об ошибке в работе конструктора, в надежде на проверку ее значения пользователем.

3. Не выполнять инициализацию в конструкторах, оставив эту работу для инициализирующих функций-членов, вызываемых пользователем до первого использования объекта.
4. Пометить объект как «непроинициализированный» и при первом вызове функции-члена для этого объекта выполнить инициализацию уже в этой функции (с вытекающими последствиями в случае невозможности успешного завершения инициализации).

Механизм обработки исключений позволяет передать вовне информацию о проблемах в работе конструкторов. Например, векторный класс может защитить свои объекты от попыток затребовать слишком много памяти в момент их создания:

```
class Vector
{
public:
  class Size{ };

  enum { max=32000 };

  Vector::Vector(int sz)
  {
    if(sz<0 || max<sz) throw Size();
    //...
  }
  //...
};
```

Код, создающий объекты типа *Vector*, может перехватывать ошибки *Vector::Size*, чтобы сделать в этом случае что-нибудь осмысленное:

```
Vector* f(int i)
{
  try
  {
    Vector* p = new Vector(i);
    //...
    return p;
  }
  catch(Vector::Size)
  {
    // обработка ошибки с размером
  }
}
```

Обработчик ошибок может использовать стандартный набор основных технологий сигнализирования об ошибках и восстановления. При каждой передаче исключения в вызывающую функцию мнение о том, что же пошло не так, может различаться. Если передавать с исключением всю необходимую информацию, то объем исходных знаний об ошибке возрастает. Другими словами, главной задачей всех технологий обработки ошибок является передача информации об ошибке из точки ее возникновения в такую точку программы, где имеется достаточно сведений для выполнения удобным и надежным способом процесса восстановления системы от последствий возникновения ошибочных ситуаций.

Применение стратегии «получение ресурса есть инициализация» является самым безопасным и элегантным способом обработки ошибок в конструкторах, запрашивающих несколько ресурсов (§14.4). По сути, эта стратегия сводит работу с несколькими ресурсами к последовательному применению техники работы с единственным ресурсом.

14.4.6.1. Исключения и инициализация членов классов

Что происходит, когда инициализирующий некоторый член класса код (прямо или косвенно) генерирует исключение? По умолчанию, исключение передается туда, откуда инициирован конструктор класса этого члена. Тем не менее, сам *конструктор может перехватить исключение*, если все его *тело и список инициализации членов заключить в* ***try****-блок*. Например:

```
class X
{
  Vector v;
  // ...
public:
  X(int) ;
  // ...
};

X::X(int s)
try
: v(s)                  // инициализация v значением s
{
  // ...
}
catch(Vector::Size)     // исключения из v перехватываются здесь
{
  // ...
}
```

14.4.6.2. Исключения и копирование

Как все конструкторы, копирующий конструктор может сигнализировать о невозможности штатного исполнения генерацией исключения. Объект в этом случае не создается. Например, копирующему конструктору класса **vector** часто нужно выделять память и копировать в нее элементы (§16.3.4, §E.3.2), и это может вызвать генерацию исключения. Перед генерацией исключения копирующему конструктору нужно освободить все занятые им ресурсы. Подробное обсуждение вопросов, связанных с управлением ресурсами и обработкой исключений для контейнерных классов, представлено в §E.2 и §E.3.

Копирующее присваивание похоже на копирующий конструктор в том отношении, что тоже может занимать ресурсы и завершаться генерацией исключения. Перед генерацией исключения здесь нужно убедиться, что оба операнда операции находятся в корректном состоянии. В противном случае нарушаются требования стандартной библиотеки, и это может привести к непредсказуемому поведению (§E.2, §E.3.3).

14.4.7. Исключения в деструкторах

С точки зрения механизма обработки исключений деструктор может быть вызван двумя способами:

1. *Нормальный вызов*: в результате обычного выхода из области видимости (§10.4.3), или операции delete (§10.4.5).
2. *Вызов в процессе обработки исключения*: в процессе раскрутки стека (§14.4) механизм обработки исключений приводит к выходу из области видимости, содержащей объект с деструктором.

В последнем случае исключение не может исходить из самого деструктора, так как это считается ошибкой в работе механизма обработки исключений, приводящей к вызову ***std*::*terminate***() (§14.7). В конце концов, ни механизм обработки исключений, ни деструкторы не в состоянии в общем случае определить, каким исключением пренебречь ради обработки другого исключения. Выход из деструктора путем генерации исключения также считается нарушением требований стандартной библиотеки (§E.2).

Деструктор может обрабатывать исключения, которые генерируются вызываемыми им функциями. Например:

```
X::~X()
try
{
  f();          // может генерировать исключения
}
catch (...)
{
  // что-нибудь делаем
}
```

Функция стандартной библиотеки ***uncaught_exception***() из файла <***exception***> возвращает ***true***, если исключение было сгенерировано, но еще не было перехвачено. Это позволяет программисту предпринимать разные действия в случаях, когда объект уничтожается нормальным образом, или же в процессе раскрутки стека.

14.5. Исключения, не являющиеся ошибками

Если мы предвидим возникновение исключений и перехватываем их для обработки таким образом, что конечное поведение программы при этом не страдает, то почему нужно в таких случаях считать исключения ошибками? Только потому, что программист считает их ошибками, а механизм обработки исключений рассматривает как механизм обработки ошибок. С другой стороны, можно рассматривать механизм обработки исключений как еще одну управляющую структуру. Например:

```
void f(Queue<X>& q)
{
  try
  {
    for(;;)
```

```
    {
      X m = q.get();     // генерирует Empty, если queue пустой (empty)
      // ...
    }
  }
  catch (Queue<X>::Empty)
  {
    return;
  }
}
```

В этом примере есть свое очарование, ибо это как раз тот случай, когда невозможно определенным образом сказать, что тут ошибка, а что нет.

Обработка исключений представляет собой менее структурированный механизм управления, нежели локальные управляющие структуры типа *if* или *for*, и он показывает меньшую эффективность в случае реальной генерации исключения. Поэтому использовать такой механизм стоит лишь тогда, когда более традиционные структуры управления либо неэлегантны, либо недопустимы. Отметим также, что стандартная библиотека предоставляет очередь queue для произвольных элементов без использования исключений (§17.3.2).

Применение исключений в качестве альтернативного варианта выхода из функции является элегантным приемом, например, для выхода из функций поиска — особенно в случае глубоко рекурсивного поиска в древовидных структурах. Например:

```
void fnd(Tree* p, const string& s)
{
  if(s == p->str) throw p;          // s найдено
  if(p->left) fnd(p->left,s);
  if(p->right) fnd(p->right,s);
}

Tree* find(Tree* p, const string& s)
{
  try
  {
    fnd(p,s);
  }
  catch (Tree* q)                   // q->str==s
  {
    return q;
  }
  return 0;
}
```

Рассмотренным применением исключений не стоит перебарщивать, ибо это ведет к запутанному коду. Всегда лучше, по возможности, придерживаться традиционной трактовки — «обработка исключений есть обработка ошибок». В этом случае код явным образом разделяется на «обычный код» и «код обработки ошибок». Это сильно улучшает читаемость кода. Однако реальный мир не расслаивается на очевидные подчасти столь же простым образом (и организация программ отражает и должна отражать этот факт).

Обработка ошибок является сложной по своей сути. Все, что помогает четко определить понятие ошибки и выявить разумную стратегию ее обработки, имеет большую ценность.

14.6. Спецификация исключений

Генерация и перехват исключений воздействует на механизм, которым функция взаимодействует с другими функциями. Поэтому небесполезно использовать *спецификацию исключений* (*exception specification*), для чего нужно в объявлении функции указать набор исключений, который она может породить. Например:

```
void f(int a) throw(x2, x3);
```

Это объявление специфицирует, что функция *f*() может быть источником исключений типа *x2*, *x3* и производных от них типов, и никаких других типов исключений. Объявленная таким образом функция предоставляет своим пользователям определенные гарантии. Если во время своего выполнения она попробует произвести действия, нарушающие объявленную гарантию, такая попытка приведет к вызову стандартной функции **std::unexpected()**. По умолчанию, функция **std::unexpected()** вызывает **std::terminate()**, а та, в свою очередь, вызывает **abort()** (§9.4.1.1).

На самом деле, следующий код

```
void f() throw(x2, x3)
{
  // нечто
}
```

эквивалентен

```
void f()
try
{
  // нечто
}
catch(x2) {throw;}                    // генерируем повторно
catch(x3) {throw;}                    // генерируем повторно
catch(...) {std::unexpected();}       // unexpected() не возвращает управление
```

Важным преимуществом рассмотренной спецификации исключений является тот факт, что объявление функции видимо всем ее пользователям (является частью спецификации ее интерфейса), а определение функции — нет. Но даже если доступны все определения библиотечных функций, вряд ли нам так уж захочется в деталях знакомиться с каждым из них. К тому же, явная спецификация типов исключений в составе объявления функции намного короче варианта с подробным ручным кодированием.

Предполагается, что функция, объявленная *без спецификации исключений*, может порождать *любое исключение*. Например:

```
int f();                              // может генерировать любые исключения
```

Функция, не порождающая никаких исключений, может быть объявлена следующим образом:

```
int g() throw();                // не генерирует исключений
```

Некоторые могут подумать, что умолчательный вариант больше подходит функциям, запрещающим исключения. Такое правило потребовало бы обязательного наличия спецификации исключений практически для всех функций, вынуждало бы частую перекомпиляцию кода и препятствовало бы взаимодействию с кодом, написанным на других языках программирования. Все это заставляло бы программистов ниспровергать механизм обработки исключений и писать паразитный код для их подавления. А это создало бы ложное чувство безопасности у людей, которые не заметили подобной «подрывной деятельности».

14.6.1. Проверка спецификации исключений

Невозможно выявить абсолютно все нарушения спецификации интерфейса на этапе компиляции. Тем не менее, на этапе компиляции реально выполняется весьма существенная проверка. О спецификациях исключений можно думать как о предположении, что функция сгенерирует все указанные в ней исключения. Простые нелепости в связи со спецификацией исключений легко выявляются в процессе компиляции.

Если некоторое объявление функции содержит спецификацию исключений, то тогда и все другие ее объявления (включая определение) обязаны иметь точно такую же спецификацию. Например:

```
int f() throw(std::bad_alloc);
int f()                         //error: отсутствие спецификации исключений
{
  //...
}
```

Важно, что не требуется проверять спецификации исключений за пределами единицы трансляции. Конкретные реализации могут это делать. Но для больших и долгоживущих программных систем они этого делать не должны, или если уж делают, то тогда им следует детектировать лишь грубейшие ошибки, которые невозможно перехватить на этапе выполнения. Смысл состоит в том, чтобы добавление исключений в каком-либо месте программы не требовало бы тотального обновления спецификаций и перекомпиляции остальных частей программы. Система может функционировать в частично обновленном состоянии, полагаясь на динамическое обнаружение неожиданных исключений на этапе выполнения. Все это очень важно для поддержки и сопровождения больших программных систем, значительные модификации которых весьма дороги и не весь их исходный код доступен.

Виртуальная функция может замещаться функцией с не менее ограничительной спецификацией исключений. Например:

```
class B
{
public:
  virtual void f();              // может генерировать любые исключения
  virtual void g() throw(X, Y);
  virtual void h() throw(X);
};
```

```
class D : public B
{
  void f() throw(X);        // ok
  void g() throw(X);        // ok: D::g() более ограничительно, чем B::g()
  void h() throw(X, Y);     // error: D::h() менее ограничительно, чем B::h()
};
```

Это правило соответствует здравому смыслу. Если в функции производного класса допускается исключение, не объявленное в соответствующей версии базового класса, вызывающая функция (клиент) может не ожидать его. С другой стороны, замещающая виртуальная функция со спецификаций, допускающей меньшее число исключений, не противоречит спецификации исключений замещенной функции.

Аналогично, вы можете присвоить значение указателя на функцию (то есть адрес функции) с более ограничительной спецификацией исключений указателю на функцию с более широкой спецификацией исключений, но не наоборот. Например:

```
void f() throw(X);
void (*pf1)() throw(X, Y) = &f;   // ok
void (*pf2)() throw() = &f;       // error: f() менее ограничительна, чем pf2
```

В частности, нельзя присваивать значение указателя на функцию с отсутствующей спецификацией исключений указателю на функцию с явно указанной спецификацией исключений:

```
void g();                       // может генерировать любые исключения
void (*pf3)() throw(X) = &g;    // error: g() менее ограничительна, чем pf3
```

Спецификация исключений не является частью типа функции, и в операторе *typedef* она не допускается. Например:

```
typedef void (*PF)() throw(X);   // error
```

14.6.2. Неожиданные исключения

Спецификация исключений может приводить к вызову *unexpected*(). В принципе, такие вызовы нежелательны, кроме как на этапе тестирования. Их можно избежать посредством тщательной организации исключений и спецификаций интерфейсов. Кроме того, вызовы *unexpected*() могут перехватываться таким образом, что они становятся безвредными.

Исключения хорошо сформированной подсистемы *Y* часто наследуют от одного класса *Yerr*. Например, при наличии определения

```
class Some_Yerr : public Yerr { /* ... */ };
```

функция, объявленная как

```
void f() throw (Xerr, Yerr, exception);
```

будет передавать любое исключений *Yerr* вызывающей функции. В частности, функция *f*() будет обрабатывать любое исключение типа *Some_Yerr*, передавая его вызывающей функции. В итоге, никакое исключение семейства Yerr не спровоцирует в *f*() вызов *unexpected*().

Все исключения стандартной библиотеки наследуются от класса *exception* (§14.10).

14.6.3. Отображение исключений

Политика завершения работы программы при обнаружении неожиданного исключения иногда является неоправданно жесткой. В таких случаях поведение *unexpected*() должно быть изменено с тем, чтобы добиться более приемлемых результатов.

Проще всего это можно сделать добавлением стандартного исключения *std::bad_exception* к спецификации исключений. В этом случае *unexpected*() будет просто генерировать *std::bad_exception*, а не вызывать функции для обработки проблемы. Например:

```
class X{ };
class Y{ };
void f() throw (X, std::bad_exception)
{
  //...
  throw Y();     // генерация "bad" exception
}
```

Спецификация исключений зафиксирует неприемлемое исключение *Y* и в ответ сгенерирует исключение типа *std::bad_exception*.

Нет ничего плохого в исключении *std::bad_exception*; оно просто запускает менее радикальный механизм реакции, чем вызов *terminate*(). Механизм этот, правда, все-таки еще сыроват: например, информация о том, какое исключение вызвало проблему, теряется.

14.6.3.1. Отображение исключений пользователем

Рассмотрим функцию *g*(), написанную для несетевого окружения. Предположим также, что функция *g*() была объявлена со спецификацией исключений, допускающей лишь исключения, относящиеся к «подсистеме *Y*»:

```
void g() throw (Yerr);
```

А теперь предположим, что нам нужно вызвать *g*() в сетевом окружении.

Естественно, что *g*() ничего не знает о сетевых исключениях и будет вызывать *unexpected*() в ответ на возникновение последних. Для успешного применения функции *g*() в сетевом (распределенном) окружении мы будем вынуждены либо написать код для обработки сетевых исключений, либо переписать *g*(). Полагая переписывание функции *g*() нежелательным и неприемлемым решением, мы предоставим собственную функцию, вызываемую вместо *unexpected*().

Как мы уже знаем, ситуация с исчерпанием памяти обрабатывается с помощью указателя *_new_handler*, который регистрируется функцией *set_new_handler*(). Аналогично, реакция на неожиданное исключение определяется указателем *_unexpected_handler*, регистрируемым функцией *std::set_unexpected*() из файла <*exception*>:

```
typedef void (*unexpected_handler) ();
unexpected_handler set_unexpected (unexpected_handler);
```

Чтобы обрабатывать неожиданные исключения так, как это нас устраивает, сначала определяем класс, позволяющий реализовать стратегию «получение ресурса есть инициализация» в отношении функций типа *unexpected*():

```
class STC    // класс для хранения и восстановления
{
  unexpected_handler old;

public:
  STC(unexpected_handler f) {old=set_unexpected(f);}
  ~STC() {set_unexpected(old);}
};
```

Затем определяем функцию ***throwY***(), которую мы хотим использовать вместо ***unexpected***() в нашем случае:

```
class Yunexpected : public Yerr { };
void throwY() throw (Yunexpected) {throw Yunexpected();}
```

Будучи использованной вместо ***unexpected***(), функция ***throwY***() отображает любое неожиданное исключение в ***Yunexpected***.

Наконец, мы предоставляем версию функции ***g***(), призванную работать в сетевой среде:

```
void networked_g() throw (Yerr)
{
  STC xx(&throwY);    // теперь unexpected() генерирует Yunexpected
  g();
}
```

Поскольку ***Yunexpected*** является производным исключением от ***Yerr***, то спецификация исключений не нарушается. Если бы ***throwY***() генерировала исключение, нарушающее спецификацию исключений, вызывалась бы функция ***terminate***().

Сохраняя и затем восстанавливая значение указателя ***_unexpected_handler***, мы предоставляем разным подсистемам возможность по-разному обрабатывать неожиданные исключения, не входя в противоречие друг с другом. В общем и целом, техника отображения неожиданных исключений в известные исключения является более гибким решением, нежели предлагаемое системой стандартное отображение в ***std::bad_exception***.

14.6.3.2. Восстановление типа исключения

Отображение неожиданного исключения в ***Yunexpected*** позволит пользователю ***networked_g***() узнать только то, что некоторое неожиданное исключение отображено в ***Yunexpected***. Пользователь не будет знать, какое именно неожиданное исключение отображено таким образом. Эта информация теряется в функции ***throwY***(). Простой метод позволяет сохранить и передать эту информацию. Например, можно следующим образом собрать информацию об исключениях ***Network_exception***:

```
class Yunexpected : public Yerr
{
public:
  Network_exception* pe;
  Yunexpected(Network_exception* p) : pe(p?p->clone():0) { }
  ~Yunexpected() {delete p;}
};
```

```
void throwY() throw(Yunexpected)
{
  try
  {
    throw;                    // повторно сгенерировать и тут же перехватить!
  }
  catch(Network_exception& p)
  {
    throw Yunexpected(&p);    // генерация отображенного исключения
  }
  catch(...)
  {
    throw Yunexpected(0);
  }
}
```

Повторная генерация и перехват исключения позволяют нам управлять любым исключением, имя которого нам известно. Функция ***throwY***() вызывается из ***unexpected***(), вызов которой концептуально порождается из обработчика ***catch***(...), так что некоторое исключение в самом деле существует и его можно генерировать повторно оператором ***throw***. При этом функция ***unexpected***() проигнорировать исключение и просто вернуть управление не может — попытайся она это сделать, в ней самой же будет сгенерировано исключение ***std::bad_exception*** (§14.6.3).

Функция ***clone***() используется для размещения копии исключения в свободной памяти. Эта копия будет существовать и после того, как произойдет очистка локальных переменных обработчика исключений.

14.7. Неперехваченные исключения

Если исключение генерируется, но не перехватывается, то вызывается функция ***std::terminate***(). Эта функция вызывается также в случаях, когда механизм обработки исключений сталкивается с повреждением стека, или когда деструктор, вызванный в процессе раскрутки стека в связи с исключением, пытается завершить работу генерацией исключения.

Как мы знаем, реакция на неожиданное исключение определяется указателем ***_unexpected_handler***, регистрируемым функцией ***std::set_unexpected***(). Аналогично, реакция на неперехваченное исключение определяется указателем ***_uncaught_handler***, регистрируемым функцией ***std::set_terminate***() из файла <***exception***>:

```
typedef void(*terminate_handler)();
terminate_handler set_terminate(terminate_handler);
```

Возвращаемым значением является указатель на прежнюю функцию, переданную для регистрации ***set_terminate***().

Функция ***terminate***() нужна там, где нужно сменить механизм обработки исключений на что-нибудь менее тонкое. Например, функция ***terminate***() могла бы использоваться для прекращения процесса и, возможно, для реинициализации систе-

мы. Функция *terminate*() предназначена для принятия решительных мер в тот момент, когда стратегия восстановления системы с помощью механизма обработки исключений потерпела неудачу и пришло время для перехода на другой уровень борьбы с ошибками.

По умолчанию *terminate*() вызывает функцию *abort*() (§9.4.1.1), что подходит в большинстве случаев, особенно во время отладки.

Предполагается, что обработчик, адресуемый указателем *_uncaught_handler*, не возвращает управление вызвавшему его коду. Если он попытается это сделать, *terminate*() вызовет *abort*().

Отметим, что вызов *abort*() означает ненормальный выход из программы. Для выхода из программы можно также использовать функцию *exit*(), имеющую возврат, который указывает системе, был ли выход нормальным или ненормальным (§9.4.1.1).

Вызываются ли деструкторы при завершении работы программы из-за неперехваченных исключений или нет, зависит от конкретных реализаций. Для некоторых систем важно, чтобы деструкторы не вызывались, так как в этом случае возможно возобновление работы программы в среде отладчика. Для других систем почти невозможно не вызывать деструкторы в процессе поиска обработчика.

Если вы хотите гарантировать корректную очистку для неперехватываемых исключений, нужно добавить обработчик *catch*(...) (§14.3.2) к телу функции *main*() помимо иных обработчиков исключений, которые вы намерены обрабатывать. Например:

```
int main ()
try
{
  // ...
}
catch (std::range_error)
{
  cerr<< "range error: Not again!\n";
}
catch (std::bad_alloc)
{
  cerr<< "new ran out of memory\n";
}
catch (...)
{
  // ...
}
```

Эта конструкция перехватит любое исключение кроме тех, которые генерируются при конструировании или уничтожении глобальных объектов. Невозможно перехватить исключение, генерируемое в процессе инициализации глобального объекта. Единственный способ получить контроль при генерации исключений в этих случаях — использовать *set_unexpected*() (§14.6.2). Это еще одна причина по возможности избегать глобальных переменных.

При перехвате исключения истинная точка программы, в которой оно было сгенерировано, неизвестна. Это меньшая информация о программе по сравнению

с тем, что о ней может знать специальная программа-отладчик. Поэтому в некоторых средах разработки для некоторых программ может оказаться предпочтительным *не* перехватывать исключения, восстановление от которых не предусматривается дизайном программы.

14.8. Исключения и эффективность

В принципе, обработку исключений можно организовать таким образом, что если исключение не сгенерировано, то нет и дополнительных накладных расходов. Кроме того, обработку исключений можно организовать таким образом, что она окажется не дороже вызова функции. Можно добиться этого без использования заметных дополнительных объемов памяти, а также соблюдая последовательность вызовов языка С и требования отладчиков. Это конечно не легко, но и альтернативы исключениям тоже не бесплатны — часто можно встретить традиционный код, половина объема которого занята обработкой ошибок. Рассмотрим простую функцию *f*(), которая на первый взгляд не имеет никакого отношения к обработке исключений:

```
void g(int);
void f()
{
  string s;
  // ...
  g(1);
  g(2);
}
```

Однако функция *g*() может генерировать исключение, и *f*() должна обеспечить корректное уничтожение строки s в этом случае. Если бы *g*() не генерировала исключений, а использовала бы иной механизм для информирования об ошибках, то приведенная выше простая функция *f*() превратилась бы в нечто вроде следующего примера:

```
bool g(int);

bool f()
{
  string s;
  // ...
  if(g(1))
    if(g(2))
      return true;
    else
      return false;
  else
    return false;
}
```

Обычно программисты не столь систематически обрабатывают ошибки, да это и не всегда нужно. В то же время, когда систематичность в обработке ошибок нужна, лучше поручить всю эту «хозяйственную работу» компьютеру, то есть механизму обработки исключений.

Спецификации исключений (§14.6) особо полезны для генерации качественного кода. Если бы мы явно указали, что функция *g*() не может генерировать исключений:

```
void g(int) throw();
```

качество машинного кода для функции *f*() значительно возросло бы. Стоит отметить, что ни одна традиционная библиотечная функция языка С не генерирует исключений, так что их можно практически в любой программе объявлять с пустой спецификацией исключений ***throw***(). В конкретных реализациях лишь немногочисленные функции, такие как ***atexit***() и ***qsort***(), могут генерировать исключения. Зная эти факты, конкретные реализации могут порождать более эффективный и качественный машинный код.

Перед тем, как приписать «С функции» пустую спецификацию исключений, подумайте, не может ли она все-таки генерировать исключения (прямо или косвенно). Например, такая функция могла быть модифицирована с применением операции new языка С++, которая генерирует исключение ***bad_alloc***, или она использует средства стандартной библиотеки языка С++, генерирующие исключения.

14.9. Альтернативы обработке ошибок

Механизм обработки исключений призван обеспечить передачу сообщений об «исключительных обстоятельствах» из одной части программы в другую. Предполагается также, что указанные части программы написаны независимо, и что та часть, которая обрабатывает исключение, может сделать что-либо осмысленное с возникшей ошибкой.

Чтобы использовать обработчики эффективно, нужна целостная стратегия. То есть различные части программы должны договориться о том, как используются исключения и где обрабатываются ошибки. Механизм обработки исключений не локален по своей сути, и поэтому следование общей стратегии очень важно. Это подразумевает, что рассмотрение стратегии обработки ошибок лучше запланировать на самые ранние стадии проектирования. Также важно, чтобы стратегия была простой (по сравнению со сложностью самой программы) и явной. Никто не будет придерживаться сложной стратегии в таких и без того сложных вопросах, как восстановление системы после возникновения в ней динамических ошибок.

Перво-наперво, нужно отказаться от идеи, что все типы ошибок можно обработать с помощью единственной технологии — это привело бы к излишней сложности. Успешные, устойчивые к ошибкам системы обычно многоуровневые. Каждый уровень разбирается с теми ошибками, которые он способен переварить, а остальные ошибки оставляет вышестоящим уровням. Предназначение ***terminate***() состоит в том, чтобы работать в случаях, когда сам механизм обработки исключений испортился, или когда он задействован не полностью и оставляет некоторые исключения неперехваченными. Аналогичным образом, ***unexpected***() предоставляет выход в тех случаях, когда отказывает защитный механизм спецификации исключений.

Не каждую функцию следует защищать «по полной программе». В большинстве систем невозможно снабдить каждую функцию достаточным числом проверок, гарантирующих либо ее успешное завершение, либо строго определенное поведение

в случае невозможности выполнить штатные вычисления. Причины, по которым полная стратегия защиты неосуществима, варьируются от программы к программе и от программиста к программисту. Тем не менее, для больших программ типично следующее:

1. Объем работы, обеспечивающий предельную защищенность кода, слишком велик, чтобы его можно было выполнять последовательно и единообразно.
2. Накладные расходы на память и быстродействие неприемлемо велики (вероятна тенденция повторять одни и те же проверки, например корректность аргументов, снова и снова).
3. Функции, написанные на других языках, скорее всего выпадут из установленных правил.
4. Локальное достижение сверхнадежности приводит к сложностям, от которых страдает надежность совокупной системы.

И тем не менее, разбиение программ на подсистемы, которые либо завершаются успешно, либо терпят неудачу определенным образом, очень важно, достижимо и экономично. Так, важная библиотека, подсистема или ключевая функция должны разрабатываться подобным образом. А спецификации исключений участвуют в формировании интерфейсов к таким библиотекам и подсистемам.

Обычно нам не приходится писать код большой системы от самого нуля и до конца. Поэтому, для реализации общей стратегии обработки ошибок во всех частях программы нам придется принять во внимание фрагменты, написанные в соответствии с иными стратегиями. При этом мы столкнемся с большим разнообразием в подходах по управлению ресурсами и вынуждены будет обращать внимание на то, в каком состоянии оставляются фрагменты после обработки ошибок. Конечная цель состоит в том, чтобы заставить работать фрагмент так, как будто он внешне следует общей стратегии обработки ошибок (хотя внутренне и подчиняется своей собственной уникальной стратегии).

Иногда возникает необходимость в смене стиля обработки ошибок. Например, после вызова традиционной библиотечной функции языка C мы можем проверять переменную errno и, возможно, генерировать исключение, или же действовать противоположным образом: перехватывать исключение и устанавливать значение errno перед возвратом в C-функцию из библиотеки C++:

```
void callC() throw(C_blewit)
{
  errno=0;
  c_function();
  if(errno)
  {
    // очистка(cleanup), если возможна и необходима
    throw C_blewit(errno);
  }
}

extern "C" void call_from_C() throw()
{
  try
```

```
  {
    c_plus_plus_function ( ) ;
  }
  catch ( . . . )
  {
    // очистка(cleanup), если возможна и необходима
    errno=E_CPLPLFCTBLEWIT;
  }
}
```

В таких случаях важно быть достаточно последовательным, чтобы гарантировать переход на некоторый единый стиль обработки ошибок.

Обработку ошибок следует, насколько это возможно, выполнять иерархически. Если функция наталкивается на ошибочную ситуацию в процессе своего выполнения, ей не следует за помощью обращаться к вызвавшей функции с тем, чтобы та справилась с проблемой восстановления и повторного выделения ресурсов — это порождает в системе циклические зависимости. К тому же, это делает программы запутанными и допускает возможность входа в бесконечные циклы обработки ошибок и восстановления.

Упрощающие методики, такие как «получение ресурса есть инициализация» и «исключение представляет ошибку» делают код обработки ошибок более регулярным. В §24.3.7.1 изложены идеи приложения инвариантов и утверждений (assertions) к механизму исключений.

14.10. Стандартные исключения

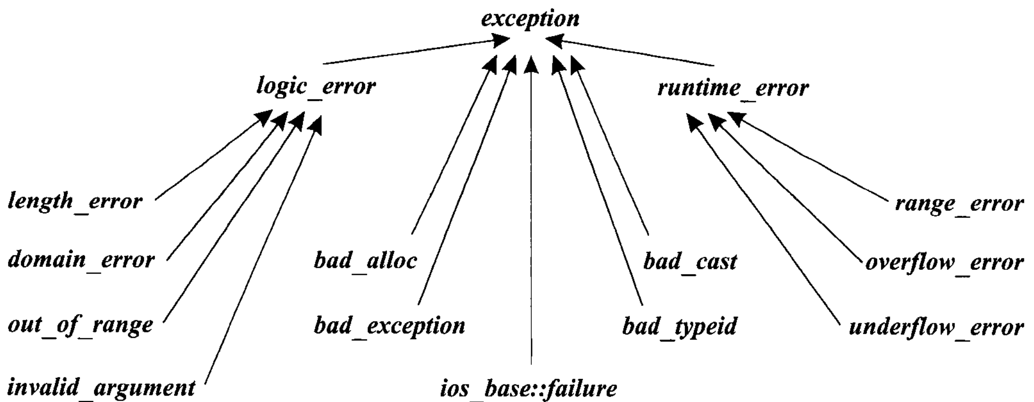
Приведем таблицу стандартных исключений, а также функций, операций и общих средств, генерирующих эти исключения:

Стандартные исключения (генерируются языком)			
Имя	**Чем генерируется**	**Ссылки**	**Загол. файл**
bad_alloc	*new*	§6.2.6.2, §19.4.5	<new>
bad_cast	*dynamic_cast*	§15.4.1.1	<typeinfo>
bad_typeid	*typeid*	§15.4.4	<typeinfo>
bad_exception	*спецификация исключений*	§14.6.3	<exception>
out_of_range	*at()*	§3.7.2, §16.3.3, §20.3.3	<stdexcept>
	bitset<>::operator[]()	§17.5.3	<stdexcept>
invalid_argument	*конструктор bitset*	§17.5.3.1	<stdexcept>
overflow_error	*bitset<>::to_ulong()*	§17.5.3.3	<stdexcept>
ios_base::failure	*ios_base::clear()*	§21.3.6	<ios>

Библиотечные исключения являются частью иерархии классов с корневым классом ***exception***, представленным в файле <***exception***>:

```
class exception
{
public:
  exception() throw();
  exception(const exception&) throw();
  exception& operator=(const exception&) throw();
  virtual ~exception() throw();

  virtual const char* what() const throw();

private:
  // ...
};
```

Графически эта иерархия выглядит следующим образом:



Такая организация кажется избыточной для восьми стандартных исключений. Однако она образует каркас для более широкой совокупности, нежели исключения стандартной библиотеки. Логическими называются ошибки (*logic_error*), которые в принципе можно выявить до начала исполнения программы или путем проверки аргументов функций и конструкторов. Ошибками времени выполнения (*runtime_error*) являются все остальные ошибки. Многие люди считают представленную схему полезной основой для всех ошибок и исключений. Я так не считаю.

Классы исключений стандартной библиотеки не добавляют функций к набору, предоставляемому корневым классом *exception*; они лишь надлежащим образом определяют необходимые виртуальные функции. Так, мы можем написать:

```
void f()
try
{
  // используем стандартную библиотеку
}
catch(exception& e)
{
  cout<< "standard library exception" << e.what() << '\n';
  // ...
}
```

```
catch ( . . . )
{
  cout << "other exception\n" ;
  // ...
}
```

Стандартные исключения являются производными от ***exception***. Но это неверно для всех исключений вообще, так что будет ошибкой пытаться перехватить все исключения с помощью одного лишь обработчика ***exception***. Также ошибкой будет полагать, что любое производное от ***exception*** исключение является исключением стандартной библиотеки: программисты могут добавлять свои собственные исключения к иерархии ***exception***.

Обратите внимание на то, что операции классов иерархии ***exception*** сами не генерируют исключений. Из этого, в частности, следует, что генерация исключений стандартной библиотеки не порождает исключения ***bad_alloc***. Механизм обработки исключений резервирует некоторое количество памяти для своей работы (исключения хранятся, скорее всего, в стеке), так что в принципе можно написать код, который постепенно исчерпает всю память. Например, ниже приведена функция, которая проверяет, кто первый исчерпает всю память — функциональный вызов или механизм обработки исключений:

```
void perverted ()
{
  try
  {
    throw exception () ;        // рекурсивная генерация исключения
  }
  catch (exception& e)
  {
    perverted () ;              // рекурсивный вызов функции
    cout << e.what () ;
  }
}
```

Цель применения здесь операции вывода состоит единственно в том, чтобы помешать компилятору использовать одну и ту же память под исключения с именем ***e***.

14.11. Советы

1. Используйте исключения для обработки ошибок; §14.1, §14.5, §14.9.
2. Не используйте исключения там, где достаточно локальных управляющих структур; §14.1.
3. Для управления ресурсами используйте стратегию «получение ресурса есть инициализация»; §14.4.
4. Не каждая программа должна безопасно обрабатывать исключения; §14.4.3.
5. Для поддержки инвариантов используйте обработчики исключений и технику «получение ресурса есть инициализация»; §14.3.2.

6. Минимизируйте использование *try*-блоков. Вместо применения явного кода обработчиков используйте технику «получение ресурса есть инициализация»; §14.4.
7. Не в каждой функция нужно обрабатывать все ошибки; §14.9.
8. Для сигнализации об ошибке в конструкторе генерируйте исключение; §14.4.6.
9. Перед генерацией исключения в операции присваивания убедитесь, что операнды находятся в корректном состоянии; §14.4.6.2.
10. Избегайте генерации исключений из деструкторов; §14.4.7.
11. Заставляйте функцию *main*() перехватывать все исключения и сообщать о них; §14.7.
12. Отделяйте обычный код от кода обработки ошибок; §14.4.5, §14.5.
13. Перед генерацией исключения в конструкторе убедитесь, что все ранее затребованные конструктором ресурсы освобождены; §14.4.
14. Организуйте управление ресурсами иерархически; §14.4.
15. Для важных интерфейсов применяйте спецификацию исключений; §14.9.
16. Не забывайте про возможные утечки памяти, выделенной операцией *new*, и неосвобожденной в случае генерации исключений; §14.4.1, §14.4.2, §14.4.4.
17. Предполагайте, что каждое исключение, которое может быть сгенерировано в функции, будет сгенерировано; §14.6.
18. Не предполагайте, что любое исключение наследует от *exception*; §14.10.
19. Библиотеки не должны безусловно завершать выполнение программ — им следует генерировать исключения и оставлять принятие решения вызывающей функции; §14.1.
20. Библиотека не должна выдавать диагностические сообщения, рассчитанные на конечного пользователя. Им нужно генерировать исключение и оставить остальную работу вызывающей функции; §14.1.
21. Разрабатывайте стратегию обработки ошибок на ранних этапах проектирования; §14.9.

14.12. Упражнения

1. (*2) Обобщите класс *STC* (§14.6.3.1) до шаблона, который использует стратегию «получение ресурса есть инициализация» для хранения и восстановления функций разных типов.
2. (*3). Завершите класс *Ptr_to_T* из §11.11 в виде шаблона, который использует исключения для сигнализации об ошибках времени выполнения.
3. (*3) Напишите функцию, которая выполняет поиск в бинарном дереве с узлами, содержащими *char**. Если находится узел, содержащий *hello*, то вызов *find*("*hello*") возвращает указатель на этот узел. Для индикации того факта, что никакой узел не найден, используйте исключение.

4. (*3) Определите класс ***Int***, который ведет себя как встроенный тип ***int***, только он еще генерирует исключения в случаях переполнения (или потери точности).
5. (*2.5) Возьмите базовые операции открытия, закрытия, чтения и записи из стандартного интерфейса языка С на вашей системе и напишите эквивалентные функции на С++, которые вызывают функции языка С, но в случае возникновения ошибок генерируют исключения.
6. (*2.5) Напишите полный шаблон ***Vector*** с исключениями типа ***Range*** и ***Size***.
7. (*1) Напишите цикл, который суммирует объекты типа ***Vector*** из предыдущего упражнения без проверки размера векторов. Почему это плохая идея?
8. (*2.5) Рассмотрите использование класса ***Exception*** как корневого для всей иерархии исключений. Как это будет выглядеть? Как это нужно использовать? К чему это может привести? Какие недостатки могут вытекать из такого подхода?
9. (*1) Имеется

   ```
   int main () { /* ... */ }
   ```

   Внесите сюда изменения, направленные на перехват всех исключений, при возникновении которых выдаются сообщения об ошибке и вызывается ***abort***(). Подсказка: функция ***call_from_C***() из §14.9 не полностью обрабатывает все случаи.
10. (*2) Напишите класс или шаблон, подходящие для реализации обратных вызовов.
11. (*2.5) Напишите класс ***Lock*** (блокировка) для некоторой системы с параллельным выполнением.

15

Иерархии классов

Абстракция — это выборочное невежество.
— Эндрю Кениг

Множественное наследование — разрешение неоднозначности — наследование и ***using**-объявление* — повторяющиеся базовые классы — виртуальные базовые классы — использование множественного наследования — управление доступом — защищенные члены — доступ к базовым классам — механизм RTTI — операция ***dynamic_cast*** — статическое и динамическое приведения типов — приведение из виртуального базового класса — операция ***typeid*** — расширенная информация о типе — правильное и неправильное применение RTTI — указатели на члены классов— свободная память — «виртуальные конструкторы» — советы — упражнения.

15.1. Введение и обзор

В этой главе обсуждается, как производные классы и виртуальные функции взаимодействуют с другими средствами языка, такими как контроль доступа, поиск имен, управление свободной памятью, конструкторы, указатели и преобразования типов. Глава состоит из пяти разделов:

- §15.2 Множественное наследование.
- §15.3 Контроль доступа.
- §15.4 Механизм RTTI (Run-Time Type Information).
- §15.5 Указатели на члены классов.
- §15.6 Свободная память.

В общем случае, класс создается из некоторой структуры базовых классов. Ввиду того, что исторически такая структура практически всегда была древовидной, ее называют *классовой иерархией* (*class hierarchy*). Мы пытаемся проектировать классы таким образом, чтобы у пользователей не было необходимости интересоваться, каким именно образом классы сформированы. В частности, механизм виртуальных функ-

ций гарантирует, что когда мы вызываем виртуальную функцию *f*() для некоторого объекта, вызывается всегда одна и та же функция в независимости от того, какой именно класс в иерархии объявил ее. В данной главе наибольшее внимание уделяется способам композиции классовых иерархий и контролю за доступом к разным частям классов, а также средствам навигации по классовым иерархиям как на этапе компиляции, так и во время выполнения программы.

15.2. Множественное наследование

Как показано в §2.5.4 и §12.3, класс может иметь более одного непосредственного базового класса, то есть два и более классов могут указываться после двоеточия в объявлении класса. Рассмотрим задачу моделирования, в которой параллельно выполняющиеся задания представлены классом ***Task***, а сбор данных и визуализация представлены классом ***Displayed***. Далее мы можем определить класс моделируемых сущностей, например класс ***Satellite*** (спутник):

```
class Satellite:public Task, public Displayed
{
  //...
};
```

Использование более одного непосредственного базового класса принято называть *множественным наследованием* (*multiple inheritance*). В противоположность этому наличие единственного непосредственного базового класса называют *одиночным наследованием* (*single inheritance*).

Кроме операций, определенных специально для ***Satellite***, мы можем использовать объединение операций классов ***Task*** и ***Displayed***. Например:

```
void f(Satellite& s)
{
  s.draw();           // Displayed::draw()
  s.delay(10);        // Task::delay()
  s.transmit();       // Satellite::transmit()
}
```

Аналогичным образом, функциям можно передавать ***Satellite*** вместо ожидаемых ими ***Task*** или ***Displayed***. Например:

```
void highlight(Displayed*);
void suspend(Task*);

void g(Satellite* p)
{
  highlight(p);       // передать указатель на Displayed-часть Satellite
  suspend(p);         // передать указатель на Task-часть Satellite
}
```

Реализация этого механизма предполагает несложные приемы от компилятора, которые обеспечивают видимость разных частей ***Satellite*** в функциях, ожидающих ***Task*** или ***Displayed***, соответственно.

Виртуальные функции работают как обычно. Например:

```
class Task
{
  // ...
  virtual void pending ( ) = 0;
} ;

class Displayed
{
  // ...
  virtual void draw ( ) = 0;
} ;

class Satellite : public Task, public Displayed
{
  // ...
  void pending ( ) ;      // замещаем Task::pending()
  void draw ( ) ;         // замещаем Displayed::draw()
} ;
```

Это гарантирует, что функции ***Satellite::draw*** ( ) и ***Satellite::pending*** ( ) будут вызваны для ***Satellite***, проинтерпретированного как ***Displayed*** и ***Task***, соответственно.

В случае одиночного наследования у программиста будет более ограниченный выбор для реализации классов ***Displayed***, ***Task*** и ***Satellite***. ***Satellite*** в таком случае может быть либо ***Task***, либо ***Displayed***, но не обоими сразу (если только ***Task*** не был реализован как производный класс от ***Displayed***, или наоборот). Выбор любой из этих альтернатив означает потерю гибкости.

А кому вообще может потребоваться такой вот класс ***Satellite***? На самом деле, это вполне реальный пример (даже если кому-то он показался абсолютно искусственным). Существовала (а может и сейчас еще существует) программа, построенная по схеме, примененной здесь для иллюстрации множественного наследования. Она использовалась для исследования построения коммуникационных систем, включающих спутники (***satellites***), наземные станции и т.д. В рамках этой модели можно отвечать на вопросы об интенсивности трафика, находить правильную реакцию на ситуацию выведения из строя наземной станции (например, из-за торнадо), отыскивать оптимальное соотношение между нагрузкой на спутниковую связь и связь по Земле и т.д. Такое интенсивное моделирование предполагает множество операций вывода информации и множество отладочных операций. Также, нам нужно хранить состояние объектов ***Satellite*** и их подкомпонентов для анализа, отладки и восстановления после ошибок.

15.2.1. Разрешение неоднозначности

Два базовых класса могут иметь функции-члены с совпадающими именами. Например:

```
class Task
{
  // ...
  virtual debug_info* get_debug ( ) ;
} ;
```

```
class Displayed
{
  // ...
  virtual debug_info* get_debug () ;
} ;
```

Для объектов типа ***Satellite*** эти неоднозначности должны устраняться:

```
void f (Satellite* sp)
{
  debug_info* dip=sp->get_debug () ;        // error: неоднозначность
  dip = sp->Task: :get_debug () ;            // ok
  dip = sp->Displayed: :get_debug () ;       // ok
}
```

Однако явное устранение неоднозначностей довольно неуклюже, так что обычно лучше определить новую функцию в производном классе:

```
class Satellite : public Task, public Displayed
{
  // ...
  debug_info* get_debug ()      // замещаем Task::get_debug() и Displayed::get_debug()
  {
    debug_info* dip1 = Task: :get_debug () ;
    debug_info* dip2 = Displayed: :get_debug () ;
    return dip1->merge (dip2) ;
  }
}
```

Это локализует информацию о базовых классах для ***Satellite***. Так как ***Satellite: :get_debug*** () замещает функции ***get_debug*** () для обоих базовых классов, ***Satellite: :get_debug*** () вызывается при каждом вызове для объекта ***Satellite***.

Квалифицированное имя ***Telstar: :draw*** () может ссылаться либо на функцию ***draw*** (), объявленную в ***Telstar***, либо в одном из базовых классов. Например:

```
class Telstar : public Satellite
{
  // ...
  void draw ()
  {
    draw () ;                              // oops!: рекурсивный вызов
    Stellite: :draw () ;                   // находим Displayed::draw
    Displayed: :draw () ;
    Satellite: :Displayed: :draw () ;      // избыточная двойная квалификация
  }
}
```

Другими словами, если ***Satellite: :draw*** () не разрешается в одноименную функцию класса ***Satellite***, то компилятор рекурсивно осуществляет поиск в базовых классах; то есть ищет ***Task: :draw*** () и ***Displayed: :draw*** (). Если находится единственное соответствие, то оно и используется. В противном случае, ***Satellite: :draw*** () либо не найдена, либо неоднозначна.

15.2.2. Наследование и using-объявление

Разрешение перегрузки не пересекает границ областей видимости различных классов (§7.4). В частности, неоднозначности между функциями из разных базовых классов не разрешаются на основе типов аргументов.

При комбинировании существенно разных классов, таких как ***Task*** и ***Displayed*** в примере с классом ***Satellite***, сходство в названиях функций не означает сходства в их назначении. Такого рода конфликты имен часто становятся полным сюрпризом для программистов. Например:

```
class Task
{
  // ...
  void debug (double p) ;
} ;

class Displayed
{
  // ...
  void debug (int v) ;
} ;

class Satellite: public Task, public Displayed
{
  // ...
} ;

void g (Satellite* p)
{
  p->debug (1) ;  // error: неоднозначно - Displayed::debug(int) или Task::debug(double)?

  p->Task: : debug (1) ;          // ok
  p->Displayed: : debug (1) ;     // ok
}
```

А что, если совпадение имен функций в разных базовых классах было продуманным проектным решением, но пользователь хочет различать их по типам аргументов? В этом случае *объявления **using*** (§8.2.2) могут ввести обе функции в общую область видимости. Например:

```
class A
{
public:
  int f(int) ;
  char f(char) ;
  // ...
} ;

class B
{
public:
  double f(double) ;
  // ...
} ;
```

```
class AB: public A, public B
{
public:
  using A::f;
  using B::f;
  char f(char);          // скрывает A::f(char)
  AB f(AB);
};

void g(AB& ab)
{
  ab.f(1);               // A::f(int)
  ab.f('a');             // AB::f(char)
  ab.f(2.0);             // B::f(double)
  ab.f(ab);              // AB::f(AB)
}
```

Объявления ***using*** позволяют программисту сформировать набор перегруженных функций из базовых классов и производного класса. Функции, объявленные в производном классе, скрывают в противном случае доступные функции из базовых классов. Виртуальные функции базовых классов могут замещаться как обычно (§15.2.3.1).

Объявление ***using*** (§8.2.2) в определении класса должно ссылаться на члены базового класса. Вне классов *объявление* ***using*** не может ссылаться на члены класса, на его производные классы и их члены. *Директивы* ***using*** (§8.2.3) нельзя помещать в определение класса и их нельзя использовать для классов.

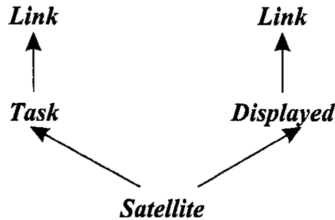
Невозможно использовать *объявления* ***using*** для получения доступа к дополнительной информации. Это лишь способ упростить доступ к имеющейся информации (§15.3.2.2).

15.2.3. Повторяющиеся базовые классы

При наличии нескольких базовых классов становится возможной ситуация, когда какой-то класс дважды окажется базовым для другого класса. Например, если бы каждый из классов ***Task*** и ***Displayed*** был бы производным от класса ***Link***, то класс ***Satellite*** получал бы бы класс ***Link*** в качестве базового дважды:

```
struct Link
{
  Link* next;
};

class Task:public Link
{
  // Link используется для списка задач типа Task (список планировщика)
  // ...
};

class Displayed: public Link
{
  // Link используется для списка отображаемых (Displayed) объектов (список показа)
  // ...
};
```

В принципе, это проблем не вызывает. Просто для представления связей используются два объекта типа ***Link***, которые не мешают друг другу. В то же время, обращаясь к членам класса ***Link***, вы рискуете нарваться на неоднозначности (§15.2.3.1). Объект класса ***Satellite*** можно изобразить графически следующим образом:



В случаях, когда базовый класс не должен быть представлен двумя раздельными объектами, нужно использовать так называемые виртуальные базовые классы (§15.2.4).

Обычно, реплицируемый базовый класс (такой как ***Link*** в нашем примере) должен быть деталью реализации и его не следует использовать вне непосредственно производных от него классов. Если к такому базовому классу нужен доступ из места, где видны его множественные копии, нужно явно квалифицировать ссылки во избежание неоднозначностей. Например:

```
void mess_with_links (Satellite* p)
{
  p->next = 0;                 // error: неоднозначность (какой Link?)
  p->Link::next = 0;           // error: неоднозначность (какой Link?)

  p->Task::next = 0;           // ok
  p->Displayed::next = 0;      // ok
  // ...
}
```

Это в точности тот же механизм, что используется для разрешения неоднозначностей при доступе к членам (§15.2.1).

15.2.3.1. Замещение

Виртуальная функция реплицируемого базового класса может замещаться (единственной) функцией производного класса. Например, можно следующим образом обеспечить объекту возможность читать свое состояние из файла и записывать его в файл:

```
class Storable
{
public:
  virtual const char* get_file () = 0;
  virtual void read () = 0;
  virtual void write () = 0;
  virtual~Storable () { }
};
```

Естественно, разрабатывая свои собственные специализированные системы, разные программисты могут воспользоваться этим определением для проектирова-

ния классов, которые могут использоваться независимо друг от друга или в некоторой комбинации. Например, в задачах моделирования при остановке работы важно сохранять состояние объектов моделирования с последующим их восстановлением при рестарте. Это можно было бы реализовать следующим образом:

```
class Transmitter : public Storable
{
public:
  void write () ;
  // ...
};

class Receiver : public Storable
{
public:
  void write () ;
  // ...
};

class Radio : public Transmitter, public Receiver
{
public:
  const char* get_file () ;
  void read () ;
  void write () ;
  // ...
};
```

Часто замещающая функция вызывает соответствующую версию базового класса, после чего выполняет специфическую именно для производного класса работу:

```
void Radio::write ()
{
  Transmitter::write () ;
  Receive::write () ;
  // далее пишем информацию, специфичную для Radio
}
```

Преобразование от типа реплицируемого базового класса к производному классу обсуждается в §15.4.2. Техника замещения каждой из функций *write*() отдельными функциями производных классов обсуждается в §25.6.

15.2.4. Виртуальные базовые классы

Пример с классом *Radio* из предыдущего раздела работает потому, что класс *Storable* реплицируется вполне безопасно, удобно и эффективно. Однако это часто не свойственно классам, служащим строительными блоками для других классов. Например, мы могли бы определить класс *Storable* так, чтобы он содержал имя файла, используемого для хранения объектов:

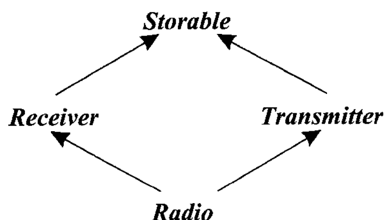
```
class Storable
{
public:
  Storable (const char* s) ;
```

```
    virtual void read () = 0;
    virtual void write () = 0;
    virtual ~Storable () {write () ; }
private:
    const char* store;

    Storable (const Storable&) ;
    Storable& operator= (const Storable&) ;
};
```

Несмотря на внешне незначительное изменение класса ***Storable***, мы должны изменить дизайн класса ***Radio***. Все части объекта должны совместно использовать единственную копию ***Storable***; в противном случае становится неоправданно сложно избегать хранения множественных копий объекта. Одним из механизмов, обеспечивающих такое совместное использование, является механизм *виртуального базового класса* (*virtual base class*). Каждый виртуальный базовый класс вносит в объект производного класса единственный (разделяемый) подобъект. Например:

```
class Transmitter : public virtual Storable
{
public:
    void write () ;
    // ...
};

class Receiver : public virtual Storable
{
public:
    void write () ;
    // ...
};

class Radio : public Transmitter, public Receiver
{
public:
    void write () ;
    // ...
};
```

Или графически:



Сравните эту диаграмму с графическим представлением объекта класса ***Satellite*** в §15.2.3, чтобы увидеть разницу между обычным и виртуальным наследованием. Любой заданный виртуальный базовый класс в иерархии наследования всегда пред-

ставим единственной копией своего объекта. В противоположность этому, *невиртуальный* базовый класс иерархии представляется своим собственным отдельным подобъектом.

15.2.4.1. Программирование виртуальных базовых классов

Определяя функции класса с виртуальным базовым классом, программист, в общем случае, не может знать, используют ли другие классы тот же самый виртуальный базовый класс. Это приводит к затруднениям, когда в рамках конкретной задачи требуется обеспечить единственный вызов функций базового класса. Например, язык гарантирует, что конструктор виртуального базового класса вызывается (явно или неявно) ровно один раз — из конструктора полного объекта (то есть из конструктора «наиболее производного класса»). Например:

```
class A                    // нет конструктора
{
  // ...
};

class B
{
public:
  B();                     // конструктор по умолчанию
  // ...
};

class C
{
public:
  C(int);                  // нет умолчательного конструктора
};

class D: virtual public A, virtual public B, virtual public C
{
  D() { /* ... */ }                 // error: для C нет умолчательного конструктора
  D(int i) : C(i) { /* ... */ };  // ok
  // ...
};
```

Конструктор виртуального базового класса выполняется до того, как начинают работать конструкторы производных классов.

Там, где это требуется, программист может имитировать такую схему работы, вызывая функции виртуального базового класса лишь из «наиболее производного класса». Например, пусть имеется класс ***Window***, знающий, как нужно отрисовывать содержимое окна:

```
class Window
{
  // ...
  virtual void draw();
};
```

Пусть также имеются классы, умеющие изменять внешний вид окна и добавлять к нему вспомогательные интерфейсные средства:

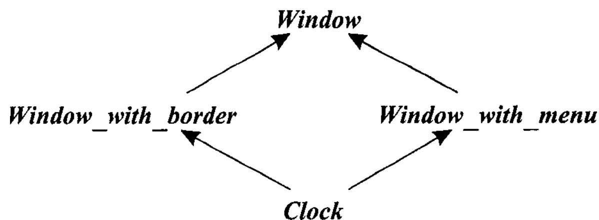
```
class Window_with_border : public virtual Window
{
 // средства для работы с рамкой
 void own_draw () ;              // отобразить рамку
 void draw () ;
};

class Window_with_menu :public virtual Window
{
 // средства для работы с меню
 void own_draw () ;              // отобразить меню
 void draw () ;
};
```

Функции ***own_draw***() не обязаны быть виртуальными, так как из вызов предполагается выполнять из виртуальных функций ***draw***(), «знающих» тип объекта, для которых они вызваны.

Далее, мы можем создать вполне правдоподобный «хронометрический» класс ***Clock***:

```
class Clock: public Window_with_border, public Window_with_menu
{
 // средства для работы с часами
 void own_draw () ;              // отобразить циферблат и стрелки
 void draw () ;
};
```

или в графическом виде:



Функции ***draw***() теперь можно написать с использованием функций ***own_draw***() так, что при вызове любых функций отрисовки функция ***Window::draw***() будет вызываться ровно один раз. И это достигается независимо от конкретного типа окна, для которого ***draw***() вызывается:

```
void Window_with_border::draw ()
{
 Window::draw () ;
 own_draw () ;                   // отобразить рамку
}

void Window_with_menu::draw ()
{
 Window::draw () ;
 own_draw () ;                   // отобразить меню
}
```

```
void Clock::draw()
{
  Window::draw();
  Window_with_border::own_draw();
  Window_with_menu::own_draw();
  own_draw();                    // отобразить циферблат и стрелки
}
```

Приведение типа от виртуального базового класса к производному обсуждается в §15.4.2.

15.2.5. Применение множественного наследования

Простейшим и наиболее очевидным применением множественного наследования является «склейка» двух несвязанных классов в качестве получения строительного блока для третьего класса. Класс ***Satellite***, который мы в §15.2 строили на основе классов ***Task*** и ***Displayed***, иллюстрирует этот случай. Такой вариант применения множественного наследования прост, эффективен и важен, но не очень интересен. В своей основе он позволяет программисту сэкономить на написании большого числа функций, переадресующих вызовы друг другу. Эта техника в целом не сильно влияет на дизайн программной системы, но может вступать в конфликт с необходимостью сокрытия деталей реализации. По счастью, чтобы технике программирования быть полезной, ей не обязательно быть слишком умной.

Применение множественного наследования для реализации абстрактных классов более фундаментально в том смысле, что влияет на проектный дизайн программы. Класс ***BB_ival_slider*** (§12.4.3) служит соответствующим примером:

```
class BB_ival_slider:
      public Ival_slider,        // интерфейс
      protected BBslider         // реализация
{
  // реализация функций Ival_slider и BBslider средствами BBslider
};
```

В этом примере два базовых класса играют различные с логической точки зрения роли. Один базовый класс формирует открытый абстрактный интерфейс, а другой является защищенным (protected) конкретным классом, предоставляющим «детали реализации». Эти роли отражаются и в стилистике классов, и в их режимах наследования (public или protected). Множественное наследование весьма существенно в данном примере, поскольку производный класс вынужден замещать виртуальные функции как «интерфейсного класса», так и «класса реализации».

Множественное наследование позволяет дочерним классам одного уровня иерархии (sibling classes) совместно использовать информацию без необходимости введения зависимости от единственного общего базового класса всей программы. Это как раз тот случай, который называют *ромбовидным наследованием* (*diamond-shaped inheritance*) (см. класс ***Radio*** в §15.2.4 и класс ***Clock*** в §15.2.4.1). Если при этом базовый класс нельзя реплицировать, его нужно использовать в иерархии как виртуальный базовый класс.

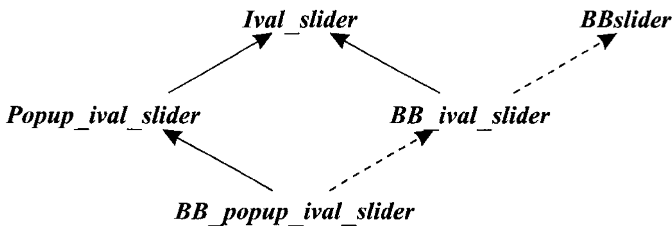
Я считаю ромбовидное наследование приемлемым в тех случаях, когда либо используется виртуальный базовый класс, либо когда непосредственно наследуемые

от него классы абстрактные. Рассмотрим снова классы семейства ***Ival_box*** из §12.4, где в конце концов я сделал классы этого семейства абстрактными, дабы отразить их чисто интерфейсную роль. Это позволило мне отнести все детали реализации в конкретные классы. Кроме того, совместное использование деталей реализации распространялось лишь на классическую иерархию классов оконной системы, в рамках которой реализация и выполнялась.

Было бы неплохо, если бы класс, реализующий более специализированный интерфейс ***Popup_ival_slider***, разделял большую часть реализации с классом, реализующим более общий интерфейс ***Ival_slider***. В конце концов, эти реализационные классы могли бы совместно использовать практически все, за исключением деталей взаимодействия с пользователем программы. В этом случае, однако, следует избегать реплицирования подобъектов ***Ival_slider*** в объектах специализированных элементов управления (ползунков). Для этого ***Ival_slider*** следует использовать как виртуальный базовый класс:

```
class BB_ival_slider : public virtual Ival_slider, protected BBslider { /* ... */ };
class Popup_ival_slider : public virtual Ival_slider { /* ... */ };
class BB_popup_ival_slider :
    public virtual Popup_ival_slider, protected BB_ival_slider { /* ... */ };
```

или графически:

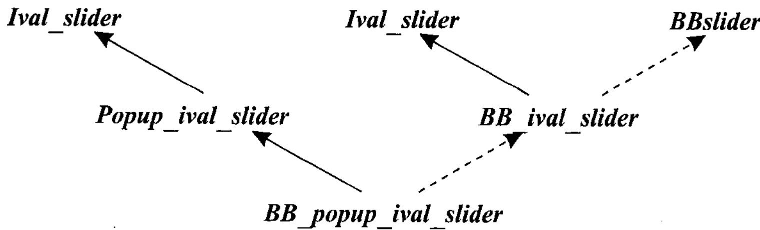


Можно легко себе представить дальнейшие интерфейсы, наследующие от ***Popup_ival_slider***, и дальнейшие реализационные классы, наследующие от этих интерфейсов и ***BB_popup_ival_slider***.

Если довести эту идею до логического конца, то все интерфейсы в системе должны наследоваться от абстрактных классов в виртуальном режиме. Такой подход выглядит наиболее логичным, общим и гибким. Но я не следую этому подходу отчасти по исторической причине, а отчасти из-за того, что наиболее распространенные практические приемы реализации концепции виртуальных базовых классов требуют избыточных затрат памяти и времени выполнения, а это делает их тотальное применение менее привлекательным. Если такие избыточные накладные расходы неприемлемы, то тогда следует обратить внимание на то, что подобъекты класса ***Ival_slider*** хранят лишь указатель на виртуальную таблицу и, как отмечалось в §15.2.4, такие абстрактные классы, не имеющие полей данных, могут реплицироваться без отрицательных последствий. Итак, мы отказываемся от виртуального наследования в пользу обычного наследования:

```
class BB_ival_slider : public Ival_slider, protected BBslider { /* ... */ };
class Popup_ival_slider : public Ival_slider { /* ... */ };
class BB_popup_ival_slider :
    public Popup_ival_slider, protected BB_ival_slider { /* ... */ };
```

или графически:



Это можно рассматривать как жизнеспособную оптимизацию более понятной альтернативы, представленной ранее. Некоторой потенциальной проблемой при этом является невозможность неявного приведения от ***BB_popup_ival_slider*** к ***Ival_slider***.

15.2.5.1. Замещение функций виртуальных базовых классов

В производных классах можно замещать виртуальные функции прямых или косвенных виртуальных базовых классов. В частности, два различных класса могут заместить разные виртуальные функции виртуального базового класса. Таким образом, несколько производных классов могут предоставить реализации интерфейса из виртуального базового класса. Пусть, например, класс ***Window*** формулирует интерфейс в виде виртуальных функций ***set_color***() и ***prompt***(). В этом случае класс ***Window_with_border*** может заместить ***set_color***() для управления цветовой схемой, а класс ***Window_with_menu*** — заместить ***prompt***() для организации интерактивного взаимодействия с пользователем:

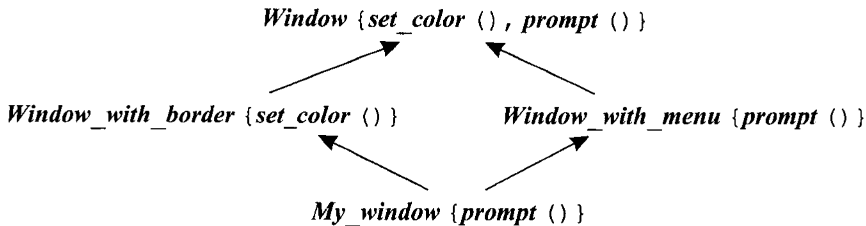
```
class Window
{
  // ...
  virtual set_color (Color) = 0;    // задать цвет фона
  virtual void prompt () = 0;
};

class Window_with_border : public virtual Window
{
  // ...
  void set_color (Color) ;          // управление цветом фона
};

class Window_with_menu : public virtual Window
{
  // ...
  void prompt () ;                  // управление взаимодействием с пользователем
};

class My_window : public Window_with_menu, public Window_with_border
{
  // ...
};
```

А что если разные производные классы заместят одну и ту же функцию? Это допускается только в случаях, когда один из таких классов наследует от всех осталь-

ных классов этой группы. То есть одна функция должна замещать все остальные. Например, класс ***My_window*** мог бы заместить ***prompt***() для улучшения возможностей, предоставляемых классом ***Window_with_menu***:

```
class My_window : public Window_with_menu, public Window_with_border
{
  //...
  void prompt () ;  // не оставляем базов. классу вопросы взаимодействия с пользователем
};
```

или графически:



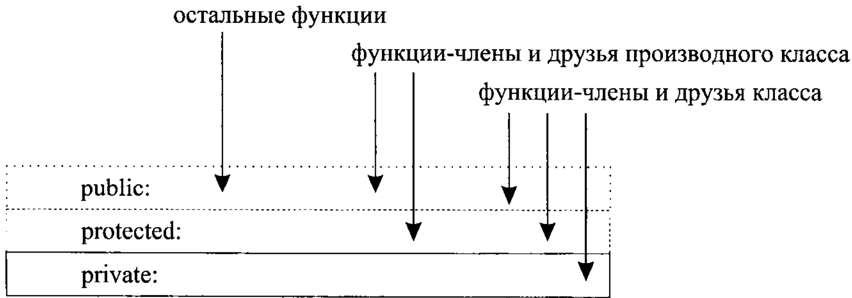
Если в двух классах замещается виртуальная функция базового класса, но при этом ни один из классов не замещает соответствующую функцию из другого, то такая классовая иерархия ошибочна: невозможно создать корректную виртуальную таблицу, ибо вызов виртуальной функции через объект «наиболее производного класса» неоднозначен. Например, если бы класс ***Radio*** из §15.2.4 не замещал функцию ***write***(), то объявления ***write***() в ***Receiver*** и ***Transmitter*** приводили бы к ошибке в определении класса ***Radio***. Такой конфликт разрешается так, как показано в этом примере — путем замещения функции в «наиболее производном классе».

15.3. Контроль доступа

Член класса может быть *закрытым* (***private***), *защищенным* (***protected***) или *открытым* (***public***):

- Если он *закрытый*, то его имя могут использовать лишь функции-члены или друзья того же самого класса.
- Если он *защищенный*, то его имя могут использовать функции-члены и друзья того же самого класса, а также функции-члены и друзья непосредственных производных классов (см. §11.5).
- Если он *открытый*, то его имя может использовать любая функция.

Данная классификация отражает точку зрения, что в отношении доступа к классу существует три вида функций: реализующие класс (друзья и члены), реализующие производный класс (друзья и члены) и остальные функции. Это можно изобразить графически следующим образом:



Контроль доступа применяется ко всем именам одинаково. То, к чему относится имя, не влияет на контроль доступа к этому имени. Это означает, что закрытыми могут быть функции-члены, типы, константы и т.д., а также поля данных. Например, классы эффективных неинтрузивных (non-intrusive — ненавязчивых) списков (§16.2.1) часто нуждаются в специальных структурах данных для учета своих элементов. Такую информацию лучше делать закрытой:

```
template<class T> class List
{
private:
  struct Link { T val; Link* next; };

  struct Chunk
  {
    enum { chunk_size = 15 };
    Link v[chunk_size];
    Chunk* next;
  };

  Chunk* allocated;
  Link* free;
  Link* get_free();
  Link* head;

public:
  class Underflow { };          // класс исключения

  void insert(T);
  T get();
  // ...
};

template<class T> void List<T>::insert(T val)
{
  Link* lnk = get_free();
  lnk->val = val;
  lnk->next = head;
  head = lnk;
}
```

```
template<class T> List<T>::Link* List<T>::get_free()
{
 if(free == 0)
 {
  // выделить новый блок памяти и включить связи в список free
 }

 Link* p = free;
 free =free->next;
 return p;
}

template<class T> T List<T>::get()
{
 if(head == 0) throw Underflow();

 Link* p= head;
 head = p->next;
 p->next =free;
 free = p;
 return p->val;
}
```

В определениях функций-членов область видимости шаблона ***List<T>*** вводится с помощью префикса ***List<T>::***, но поскольку возврат функции ***get_free()*** указывается раньше, то его нужно обозначать с полной квалификацией, то есть ***List<T>::Link***, а не просто ***Link***.

А глобальные функции (за исключением друзей) такого доступа не имеют:

```
void would_be_meddler(List<T>* p)
{
 List<T>::Link* q = 0;                  // error: List<T>::Link - закрыто

 q = p->free;                           // error: List<T>::free - закрыто
 // ...
 if(List<T>::Chunk::chunk_size >31)  // error: List<T>::Chunk::chunk_size - закрыто
 {
  //...
 }
 //...
}
```

В определении класса (когда используется ключевое слово ***class***) по умолчанию все члены закрытые, а в определении структуры (когда используется ключевое слово ***struct***) все члены по умолчанию открытые (§10.2.8).

15.3.1. Защищенные члены классов

Рассмотрим пример с иерархией классов ***Window*** из §15.2.4.1. Функции ***own_draw()*** были задуманы как строительные блоки для применения в производных классах, но их использование в остальных случаях небезопасно. А функции ***draw()***, наоборот, годятся к использованию в любых контекстах. Это различие можно точно выразить разделением общего интерфейса класса ***Window*** на защищенную (***protected***) и открытую (***public***) части:

```
class Window_with_border
{
public:
  virtual void draw () ;
  // ...
protected:
  void own_draw () ;
  // ...
private:
  // представление и т.п.
};
```

Прямой доступ к защищенным членам базового класса разрешен в производном классе лишь для объектов этого же класса:

```
class Buffer
{
protected:
  char a[128] ;
  // ...
};

class Linked_buffer : public Buffer { /* ... */ };

class Cyclic_buffer : public Buffer
{
  // ...
  void f(Linked_buffer* p)
  {
    a[0]=0;           // ok: доступ к собственному защищенному члену
    p->a[0]=0;        // error: доступ к защищенному члену другого типа
  }
};
```

Это предохраняет от довольно тонких ошибок, связанных с возможностью одного производного класса портить данные другого производного класса.

15.3.1.1. Применение защищенных членов класса

Простая модель «закрытый/открытый» для сокрытия данных хорошо работает в случае конкретных классов (§10.3). Для иерархий же классов существует два вида пользователей классов: производные классы и «простая публика». От имени этих пользователей и производят свои действия над классовыми объектами функции-члены класса и друзья. Модель «закрытый/открытый» позволяет программисту различать разработчиков классов и обычных пользователей, но она не предоставляет ничего особого в обслуживании производных классов.

Защищенные члены классов более подвержены неправильному использованию, чем закрытые члены. Часто их наличие свидетельствует об ошибках проектирования. Помещение в защищенные секции общих классов заметного объема данных, доступных производным классам, открывает возможность для порчи этих данных. Еще того хуже, защищенные данные, как и открытые, намного труднее реструктурировать ввиду сложности нахождения всех случаев их использования. А это усложняет процесс сопровождения программных систем.

По счастью, использовать защищенные данные необязательно; наиболее предпочтительным вариантом являются закрытые данные, что и имеет место по умолчанию. По моему опыту, всегда имеются альтернативы помещению изрядного количества информации в общий класс для ее непосредственного использования в производных классах.

Обратите внимание, что все это не имеет отношения к защищенным функциям — они прекрасно задают операции для применения их в производных классах. Иллюстрацией служит класс ***Ival_slider*** из §12.4.2. Если бы в этом примере функции класса были бы закрытыми, стало бы невозможно создавать дальнейшие производные классы.

Технические примеры доступа к защищенным членам приведены в §C.11.1.

15.3.2. Доступ к базовым классам

Аналогично членам класса, можно сам базовый класс объявить в процессе наследования закрытым (***private***), защищенным (***protected***) или открытым (***public***). Например:

```
class X: public B { /* ... */ };
class Y: protected B { /* ... */ };
class Z: private B { /* ... */ };
```

Открытое наследование делает производный класс подтипом базового; это наиболее распространенная форма наследования. Защищенную и закрытую формы наследования применяют для использования деталей реализации. Защищенное наследование полезно в случае классовых иерархий, в которых построение дальнейших производных классов является нормой; класс ***Ival_slider*** из §12.4.2 может служить хорошим примером. Закрытое наследование применяется для построения производных классов, закрывающих доступ к интерфейсу базового класса и обеспечивающих большие гарантии, чем базовый класс. Например, шаблон ***Vector<T*>*** добавляет проверку типа к своему базовому классу ***Vector<void*>*** (§13.5). Также, если бы нам потребовалась гарантия того, что всегда доступ к векторам проверяется (см. §3.7.2), то базовый класс для ***Vec*** пришлось бы объявить закрытым (чтобы предотвратить преобразование из ***Vec*** в непроверяемый базовый класс ***vector***):

```
template<class T> class Vec: private vector<T> { /* ... */ }; // вектор с проверкой диапазона
```

Если в определении класса опустить спецификатор режима наследования. то *для ключевого слова* ***class*** *по умолчанию режим наследования будет считаться закрытым* (***private***), а *для ключевого слова* ***struct*** *он будет считаться открытым* (***public***). Например:

```
class XX: B { /* ... */ };    // закрытое наследование от B
struct YY: B { /* ... */ };   // открытое наследование от B
```

Чтобы код был более читаемым, лучше явно использовать спецификатор для режима наследования.

Этот спецификатор управляет доступом к членам базового класса и преобразованием указателей и ссылок из типа производного класса к типу базового класса. Рассмотрим класс ***D***, производный от базового класса ***B***:

- Если ***B*** является *закрытым* базовым классом, его открытые и защищенные члены могут использоваться только функциями-членами и друзьями ***D***. Только функции-члены и друзья класса ***D*** могут преобразовывать ***D**** в ***B****.
- Если ***B*** является *защищенным* базовым классом, его открытые и защищенные члены могут использоваться только функциями-членами и друзьями ***D***, а также функциями-членами и друзьями классов, производных от ***D***. Только функции-члены и друзья класса ***D***, а также функции-члены и друзья классов, производных от ***D***, могут преобразовывать ***D**** в ***B****.
- Если ***B*** является *открытым* базовым классом, его открытые члены могут использоваться любой функцией. Кроме того, его защищенные члены могут использоваться функциями-членами и друзьями ***D***, а также функциями-членами и друзьями классов, производных от ***D***. Любая функция может преобразовывать ***D**** в ***B****.

Это, в основном, является переформулировкой правил для доступа к членам (§15.3). Мы выбираем режим наследования (режим доступа к базовым классам) по тем же соображениям, что и режим доступа к членам. Например, я предпочел сделать ***BBwindow*** защищенным базовым классом для ***Ival_slider*** (§12.4.2), потому что ***BBwindow*** служит частью реализации ***Ival_slider***, а не частью его интерфейса. Я, однако, не стал полностью закрывать ***BBwindow*** (то есть наследовать в режиме private), так как хотел оставить возможность получать классы, производные от ***Ival_slider***, которым тоже требуется доступ к реализации.

Технические примеры доступа к базовым классам приведены в §C.11.2.

15.3.2.1. Множественное наследование и контроль доступа

Если некоторое имя или базовый класс в принципе доступны по нескольким путям в рамках иерархии множественного наследования, то они достижимы лишь в случае, когда достижимы по любому из этих путей. Например:

```
struct B
{
  int m;
  static int sm;
  //...
};

class D1 : public virtual B { /* ... */ };
class D2 : public virtual B { /* ... */ };
class DD : public D1, private D2 { /* ... */ };

DD* pd = new DD;
B* pb = pd;            // ok: доступ через D1
int i1 = pd->m;        // ok: доступ через D1
```

Если некоторая единичная сущность доступна по нескольким путям, мы все равно можем обращаться к ней без возникновения неоднозначностей. Например:

```
class X1 : public B { /* ... */ };
class X2 : public B { /* ... */ };
class XX : public X1, public X2 { /* ... */ };

XX* pxx = new XX;
```

```
int i1 = pxx->m;        // error, неоднозначность: XX::X1::B::m или XX::X2::B::m
int i2 = pxx->sm;       // ok: единственный B::sm в XX
```

15.3.2.2. Множественное наследование и контроль доступа

*Объявление **using*** не может использоваться для получения доступа к дополнительной информации. Оно просто делает доступную информацию более удобной для применения. С другой стороны, если доступ к информации имеется, его можно дать и другим пользователям. Например:

```
class B
{
private:
  int a;

protected:
  int b;

public:
  int c;
};

class D: public B
{
public:
  using B::a;           // error: B::a - закрыто
  using B::b;           // делает B::b общедоступным через D
};
```

Комбинируя *объявления **using*** с закрытым или защищенным режимами наследования, можно явно открывать доступ к некоторому подмножеству членов класса. Например:

```
class BB : private B    // даем доступ к B::b и B::c, но не к B::a
{
  using B::b;
  using B::c;
};
```

См. также §15.2.2.

15.4. Механизм RTTI (Run-Time Type Information)

Вероятным использованием семейства классов ***Ival_box***, определенных в §12.4, будет передача их системе, управляющей экраном, чтобы та вернула объекты этого типа в момент наступления некоторых событий. Так работает большинство систем графических пользовательских интерфейсов. Но эти системы ничего не знают о типах семейства ***Ival_box***, так как задаются в терминах своих собственных классов и объектов, а не классов нашего приложения. В результате возникает неприятный эффект потери типа объектов, которые мы передаем системе, а потом получаем их от нее назад.

Восстановление потерянного типа объекта может заключаться в возможности как-то спросить объект о его типе. Для работы с объектами мы обычно располагаем

подходящего типа указателями или ссылками на них. Поэтому наиболее очевидной и полезной операцией над объектами на этапе выполнения программы будет операция преобразования к ожидаемому типу, которая возвращает корректное значение указателя только если такое преобразование выполнимо (или нулевой указатель в противном случае). Операция ***dynamic_cast*** как раз и предназначена для этого. Например, пусть система вызывает функцию ***my_event_handler***() с указателем на объект (окно) типа ***BBwindow***, в котором произошла какая-то активность. Затем я мог бы реагировать на событие вызовом функции ***do_something***() семейства классов ***Ival_box***:

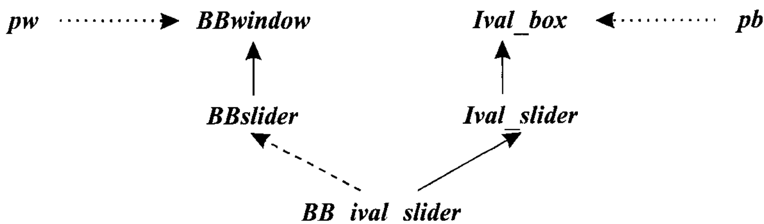
```
void my_event_handler (BBwindow* pw)
{
  if(Ival_box* pb = dynamic_cast<Ival_box*> (pw) )   // указывает pw на Ival_box?
    pb->do_something () ;
  else
  {
    // Oops! неожиданное событие
  }
}
```

Происходящее можно условно объяснить так, что ***dynamic_cast*** переводит с языка реализации системы графического интерфейса пользователя на язык нашего приложения. Очень важно отметить, что в примере отсутствует точный тип объекта. Объект лишь должен иметь тип одного из классов семейства ***Ival_box***, скажем ***Ival_slider***, реализованный одним из видов ***BBwindow***, скажем ***BBslider***. Нет никакой необходимости в знании точного типа объекта в этом взаимодействии между «системой» и приложением. Любой интерфейс существует лишь для открытия важных аспектов взаимодействия. А несущественные детали любой хороший интерфейс скрывает.

Графически, работу операции

```
pb = dynamic_cast<Ival_box*> (pw)
```

можно изобразить следующим образом:



Стрелки от ***pw*** и ***pb*** символизируют указатели на передаваемый объект, а остальные стрелки отражают отношения наследования между различными частями передаваемого объекта.

Получение и использование *информации о типе объекта на этапе выполнения программы* обычно называют механизмом *RTTI* (*run-time type information*).

Приведение типа от базового класса к производному обычно называют *понижающим приведением* (*downcast*) из-за традиции рисовать классовые иерархии свер-

ху вниз. Соответственно, приведение от производного класса к базовому называют *повышающим приведением* (*upcast*). Преобразования между классами одного уровня иерархии называют *перекрестным приведением* (*crosscast*).

15.4.1. Операция dynamic_cast

Операция ***dynamic_cast*** имеет два операнда: тип, заключенный в угловые скобки, и указатель (или ссылка), заключенный в круглые скобки.

Сначала рассмотрим вариант с указателем:

```
dynamic_cast<T*> (p)
```

Если ***p*** имеет тип ***T**** или ***D****, где ***T*** является базовым классом для ***D***, результат будет таким же, как при простом присваивании ***p*** указателю типа ***T****. Например:

```
class BB_ival_slider : public Ival_slider, protected BBslider
{
  //...
};

void f(BB_ival_slider* p)
{
  Ival_slider* pi1 = p;                              // ok
  Ival_slider* pi2 = dynamic_cast<Ival_slider*> (p) ; // ok

  BBslider* pbb1 = p;  // error: BBslider - защищенный базовый класс
  BBslider* pbb2 = dynamic_cast<BBslider*> (p) ;     // ok: pbb2 станет 0
}
```

Рассмотренный пример большого интереса не представляет. Тем не менее, все равно приятно, что операция ***dynamic_cast*** не допускает случайных нарушений уровня доступа к закрытым и защищенным базовым классам.

Свое основное предназначение операция ***dynamic_cast*** демонстрирует в случаях, когда компилятор не может выяснить, корректно преобразование типов или нет. Тогда операция

```
dynamic_cast<T*> (p)
```

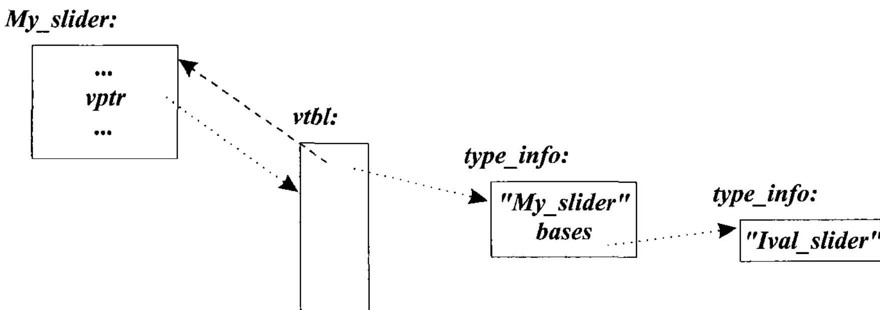
обращается к объекту, на который указывает ***p***. Если это объект типа ***T*** или типа, у которого имеется уникальный базовый класс ***T***, то операция ***dynamic_cast*** возвращает указатель типа ***T**** на этот объект; в противном случае возвращается *нуль*. Если значение ***p*** равно ***нулю***, то ***dynamic_cast<T*> (p)*** возвращает *нуль*. Обратите внимание на необходимость преобразования к уникально идентифицируемому объекту. Можно сконструировать примеры, когда преобразование невозможно и возвращается нуль из-за того, что объект, на который показывает ***p***, содержит несколько подобъектов базового класса ***T*** (см. §15.4.2).

Для выполнения понижающего или перекрестного приведения нужно, чтобы аргумент операции ***dynamic_cast*** был ссылкой или указателем на полиморфный тип. Например:

```
class My_slider:public Ival_slider // базовый класс - полиморфный (имеет виртуаль. ф-ии)
{
  //...
};
```

```
class My_date : public Date        // базовый класс - неполиморфный (нет виртуаль. ф-ий)
{
  // ...
};

void g (Ival_box* pb, Date* pd)
{
  My_slider* pd1 = dynamic_cast<My_slider*> (pb) ; // ok
  My_date*  pd2 = dynamic_cast<My_date*> (pd) ;  // error: Date - не полиморфный тип
}
```

Требование полиморфности указателя облегчает компилятору реализацию ***dynamic_cast***, ибо в этом случае легко найти место для хранения необходимой информации о типе объекта. В типичном случае специальный объект с информацией о типе прикрепляется к полиморфному объекту посредством добавления указателя на этот специальный объект к виртуальной таблице класса (§2.5.5). Например:



Пунктирная линия изображает смещение, позволяющее найти начало полного объекта при наличии лишь указателя на полиморфный подобъект. Ясно, что ***dynamic_cast*** можно реализовать весьма эффективно. Все, что для этого требуется, это несколько сравнений объектов ***type_info***, представляющих базовые классы (и никаких дорогостоящих табличных поисков или сравнений строк).

Ограничение диапазона действия операции ***dynamic_cast*** лишь полиморфными типами оправдано и с логической точки зрения. Действительно, если у объекта отсутствуют виртуальные функции, им невозможно безопасно пользоваться без знания его типа. Поэтому его нужно аккуратно применять лишь в контекстах, в которых его тип известен. А если тип объекта известен, то нет необходимости в операции ***dynamic_cast***.

Результирующий тип операции ***dynamic_cast*** не обязан быть полиморфным. Это позволяет нам «завернуть» (wrap) конкретный тип в полиморфный с целью, скажем, его передачи системе ввода/вывода (§25.4.1), а позднее «развернуть» (unwrap) этот конкретный тип. Например:

```
class Io_obj                          // базовый класс для системы ввода/вывода
{
  virtual Io_obj* clone () = 0;
};

class Io_date: public Date, public Io_obj { };
```

```
void f(Io_obj* pio)
{
  Date* pd = dynamic_cast<Date*> (pio) ;
  //...
}
```

Применение ***dynamic_cast*** для приведения к типу ***void**** можно использовать, чтобы вычислить адрес начала объекта полиморфного типа. Например:

```
void g (Ival_box* pb, Date* pd)
{
  void* pd1 = dynamic_cast<void*> (pb) ;  // ok
  void* pd2 = dynamic_cast<void*> (pd) ;  // error: Date - не полиморфный тип
}
```

Это полезно лишь при взаимодействии с очень низкоуровневыми функциями.

15.4.1.1. Применение dynamic_cast к ссылкам

Для обеспечения полиморфного поведения доступ к объекту должен выполняться через указатель или по ссылке. Когда ***dynamic_cast*** используется с указателями, возврат нуля означает невозможность приведения. Но к ссылкам все это абсолютно неприменимо.

Работая с указателями, мы всегда должны учитывать возможность его равенства нулю (это означает, что указатель не адресует никакого объекта). То есть результат работы операции ***dynamic_cast*** над указателями должен всегда проверяться явным образом, а саму операцию ***dynamic_cast***<***T****> (***p***) можно трактовать как вопрос: «Действительно ли адресуемый указателем ***p*** объект имеет тип ***T***?».

С другой стороны, мы можем твердо рассчитывать на то, что ссылка действительно указывает на некоторый объект. Следовательно, ***dynamic_cast***<***T***&> (***r***) для ссылки ***r*** рассматривается уже не как вопрос, а как утверждение: «Объект, на который ссылается ***r***, имеет тип ***T***». Результат операции ***dynamic_cast*** тестируется в этом случае самой системой, и если ссылка имеет неправильный (несовместимый с заявленным) тип, то генерируется исключение ***bad_cast***. Например:

```
void f(Ival_box* p, Ival_box& r)
{
  if(Ival_slider* is = dynamic_cast<Ival_slider*> (p) )  // указывает ли p на Ival_slider?
  {
    // используем is
  }
  else
  {
    // *p это не ползунок
  }

  Ival_slider& is = dynamic_cast<Ival_slider&> (r) ;       // r ссылается на Ival_slider!
  // используем is
}
```

Различие в поведении операции ***dynamic_cast*** в ее работе над указателями и ссылками отражает фундаментальный факт различия между самими указателями и ссылками.

Если пользователю нужен надежный код в случае, когда ***dynamic_cast*** применяется к ссылкам, нужно перехватывать и обрабатывать исключение ***bad_cast***. Например:

```
void g()
{
  try
  {
    f(new BB_ival_slider, *new BB_ival_slider); // аргументы передаются как Ival_box
    f(new BBdial, *new BBdial);                 // аргументы передаются как Ival_box
  }
  catch (bad_cast)                              // §14.10
  {
    // ...
  }
}
```

Первый вызов *f*() завершится нормально, в то время как второй вызовет исключение ***bad_cast***, которое будет перехвачено и обработано.

Явные проверки на нуль в случае работы с указателями могут быть случайно пропущены. Если это вас беспокоит, вы можете написать преобразующую функцию, которая вместо возврата нуля генерирует исключение (§15.8[1]).

15.4.2. Навигация по иерархиям классов

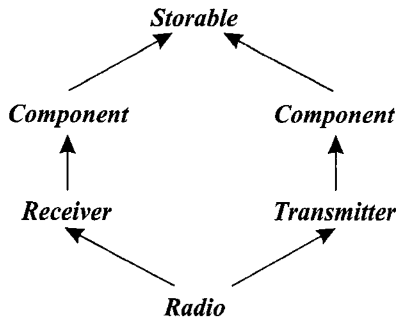
Когда используется лишь одиночное наследование, класс совместно с его базовыми классами образуют дерево с корнем в единственном базовом классе. Это просто, но часто слишком ограничительно. Когда используется множественное наследование, единственного корня не существует. Само по себе это не усложняет суть дела. Однако если класс появляется в иерархии более одного раза, нам следует проявлять осторожность при обращении к объектам такого класса.

Естественно, мы пытаемся конструировать иерархии настолько простыми, насколько нам позволяет делать это само приложение. Но если иерархия получилась нетривиальной, перед нами встает проблема навигации по этой иерархии с целью выявления класса с подходящим интерфейсом. Эта потребность проявляется в двух аспектах. Иногда нам нужно явно поименовать объект базового класса или член базового класса (см. §15.2.3 и §15.2.4.1). В иных случаях нам бывает необходимо получить указатель на объект, представляющий базовый или производный класс объекта, при наличии указателя на полный объект или некоторый подобъект (см. §15.4 и §15.4.1).

Рассмотрим способы навигации по иерархии классов с применением приведений типа для получения указателя желаемого типа. Для иллюстрации самого механизма и регулирующих его правил рассмотрим классовую иерархию, содержащую как повторяющийся базовый класс, так и виртуальный базовый класс:

```
class Component: public virtual Storable { /*...*/ };
class Receiver: public Component { /*...*/ };
class Transmitter: public Component { /*... */ };
class Radio : public Receiver, public Transmitter { /*...*/ };
```

или в графическом виде:



В этом примере объект типа ***Radio*** содержит два подобъекта класса ***Component***. Следовательно, динамическое приведение операцией ***dynamic_cast*** типа ***Storable*** в тип ***Component*** в рамках объекта ***Radio*** неоднозначно и вернет нуль. Просто нет способа узнать, какой именно ***Component*** запрашивает программист:

```
void h1 (Radio& r)
{
  Storable* ps = &r;
  // ...
  Component* pc = dynamic_cast<Component*> (ps) ;      // pc = 0
}
```

Эта неоднозначность в общем случае на этапе компиляции не выявляется:

```
void h2 (Storable* ps)      // ps может указывать на Component, а может и нет
{
  Component* pc = dynamic_cast<Component*> (ps) ;
  // ...
}
```

Такое выявления неоднозначности на этапе выполнения требуется лишь для виртуальных базовых классов. Для обычных базовых классов всегда при понижающих преобразованиях существует единственный подобъект (§15.4), в то время как для повышающих преобразований и здесь возникает аналогичная неоднозначность, обнаруживаемая на этапе выполнения.

15.4.2.1. Операции static_cast и dynamic_cast

Операция ***dynamic_cast*** может преобразовывать полиморфный виртуальный базовый класс в производный класс или в класс того же уровня иерархии («братский» класс) (§15.4.1). Операция ***static_cast*** (§6.2.7) не анализирует объект, тип которого она приводит, и поэтому она не может этого сделать:

```
void g (Radio& r)
{
  Receiver* prec = &r;                        // Receiver обычный базовый класс для Radio
  Radio* pr = static_cast<Radio*> (prec) ;    // ok, без проверки
  pr = dynamic_cast<Radio*> (prec) ;          //o k, проверка на этапе выполнения
```

```
    Storable* ps = &r;                    // Storable - виртуальный базовый класс для Radio
    pr = static_cast<Radio*> (ps) ;       // error: приведение из virtual base невозможно
    pr = dynamic_cast<Radio*> (ps) ;      // ok, проверка на этапе выполнения
}
```

Операция *dynamic_cast* требует полиморфного операнда, потому что в неполиморфном объекте нет информации, которую можно использовать для того, чтобы решить, является ли указанный класс базовым для него. Объекты типов, имеющие ограничения на размещение в памяти, накладываемые языками Fortran или C, могут использоваться в качестве виртуальных базовых классов. Для таких объектов имеется лишь статическая информация о типе. Информация, необходимая для определения типа на этапе выполнения, включает в себя информацию, необходимую для реализации операции *dynamic_cast*.

А зачем вообще может потребоваться операция *static_cast* в контексте навигации по классовым иерархиям? Ведь дополнительные накладные расходы по применению операции *dynamic_cast* невелики (§15.4.1). Однако существуют миллионы строк кода, написанные до появления *dynamic_cast*. Они опираются на альтернативные способы проверки корректности преобразований, так что проверка операцией *dynamic_cast* кажется избыточной. В типичном случае этот код использует приведения в стиле языка C (§6.2.7), и поэтому в нем остаются тонкие и трудноуловимые ошибки. В общем, где возможно используйте существенно более безопасные приведения операцией *dynamic_cast*.

Компилятор не имеет представления о том, что адресуется указателем типа *void**. Отсюда следует, что операция *dynamic_cast*, которой требуется «заглянуть в объект» для выявления его типа, не может преобразовывать тип *void**. Здесь-то и требуется *static_cast*. Например:

```
Radio* f(void* p)
{
    Storable* ps = static_cast<Storable*> (p) ;     // под ответственность программиста
    return dynamic_cast<Radio*> (ps) ;
}
```

Обе операции приведения — *dynamic_cast* и *static_cast*, учитывают модификатор *const* и уровни доступа. Например:

```
class Users: private set<Person> { /* ... */ } ;

void f(Users* pu, const Receiver* pcr)
{
    static_cast<set<Person>*> (pu) ;            // error: нарушение доступа
    dynamic_cast<set<Person>*> (pu) ;           // error: нарушение доступа

    Static_cast<Receiver*> (pcr) ;              // error: невозможно "снять" const
    dynamic_cast<Receiver*> (pcr) ;             // error: невозможно "снять" const

    Receiver* pr = const_cast< Receiver*> (pcr) ; // ok
    // ...
}
```

Невозможно осуществить преобразование к закрытому базовому классу, а для преодоления действия модификатора *const* (или *volatile*) требуется операция *const_cast* (§6.2.7). И даже в этом случае, результат надежен, лишь если объект не был с самого начала объявлен как *const* (или *volatile*) (§10.2.7.1).

15.4.3. Конструирование и уничтожение классовых объектов

Объект класса — это нечто большее, чем просто область памяти (§4.9.6). Объект класса создается поверх «сырой памяти» («raw memory») с помощью конструкторов, и эта память становится снова «сырой» после отработки деструкторов. Создание протекает снизу вверх, а уничтожение идет сверху вниз; при этом объект класса является таковым в той мере, в какой он был создан, и до того, как был уничтожен. Этот факт отражен в правилах для RTTI, обработки исключений (§14.4.7) и виртуальных функций.

Крайне неразумно полагаться на порядок создания и уничтожения объектов, но порядок этот отражается на вызовах виртуальных функций, работе операций ***dynamic_cast*** и ***typeid*** (§15.4.4) в момент, когда объект создан не полностью. Например, если конструктор класса ***Component*** из иерархии (§15.4.2) вызывает виртуальную функцию, будет вызвана версия классов ***Storable*** или ***Component***, но не версии классов ***Receiver***, ***Transmitter*** или ***Radio***. В этой фазе конструирования объект еще не является объектом ***Radio***, а является лишь частично созданным объектом. Лучше всего избегать вызова виртуальных функций на этапе создания или уничтожения объектов.

15.4.4. Операция typeid и расширенная информация о типе

Операция ***dynamic_cast*** удовлетворяет большинство потребностей в информации о типе на этапе выполнения программы. Особо важно, что она гарантирует, что использующий эту операцию код корректно работает с классами, производными от некоторого указанного программистом класса. Таким образом, ***dynamic_cast*** демонстрирует гибкость и расширяемость, свойственную виртуальным функциям.

Тем не менее, бывают случаи, когда нужно узнать точный тип объекта. Например, потребовалось узнать имя класса объекта или его расположение в памяти. Для этого имеется операция ***typeid***, возвращающая объект, содержащий имя операнда этой операции. Если бы ***typeid***() была функцией, ее объявление выглядело бы примерно следующим образом:

```
class type_info;
const type_info& typeid(type_name) throw();              // псевдообъявление
const type_info& typeid(expression) throw(bad_typeid);   // псевдообъявление
```

То есть ***typeid***() возвращает ссылку на тип из стандартной библиотеки, называемый ***type_info*** (определен в файле <***typeinfo***>). Для операндов ***type_name*** (имя типа) или ***expression*** (выражение) операцией возвращается ссылка на объект ***type_info***, содержащий собственно тип или тип выражения. Чаще всего операция ***typeid***() вызывается для объектов, адресуемых указателем или ссылкой:

```
void f(Shape& r, Shape* p)
{
  typeid(r);   // тип объекта, на который ссылается r
  typeid(*p);  // тип объекта, на который указывает p
  typeid(p);   // тип указателя (здесь это Shape*)
}
```

Если значение операнда-указателя полиморфного типа равно ***нулю***, операция ***typeid***() генерирует исключение ***bad_typeid***. Если операнд операции ***typeid***() имеет

неполиморфный тип или он не является lvalue, то результат определяется во время компиляции без вычисления выражения операнда.

Часть класса *type_info* (не зависящая от конкретной реализации) выглядит следующим образом:

```
class type_info
{
public:
  virtual ~type_info () ;                       // полиморфный

  bool operator== (const type_info&) const;     // можно сравнивать
  bool operator != (const type_info&) const;
  bool before (const type_info&) const;         // упорядочение

  const char* name () const;                    // имя типа

private:
  type_info (const type_info&) ;                // предотвращает копирование
  type_info& operator= (const type_info&) ;     // предотвращает копирование
  // ...
};
```

Функция *before*() помогает сортировать объекты типа *type_info*. Нет никакой связи между отношениями упорядочения функцией *before*() и отношениями наследования.

Не гарантируется, что есть только один объект *type_info* для каждого типа в системе. На самом деле, в системах с библиотеками динамической компоновки трудно избежать дублирования объектов *type_info*. Следовательно, нужно применять операцию == для выявления эквивалентности объектов *type_info*, а не сравнивать между собой значения указателей на такие объекты.

Иногда нам нужно знать точный тип объекта для выполнения некоторых действий с полным объектом (а не с отдельными его фрагментами базовых типов). В идеале такие действия должны выполняться виртуальными функциями, когда знать точный тип объекта нет необходимости. Но в некоторых случаях невозможно построить общий интерфейс ко всем типам объектов, и экскурс по типам объектов становится неизбежным (§15.4.4.1). Еще одним, более простым случаем является ситуация, когда имена типов нужно выводить для диагностики работы программы:

```
#include <typeinfo>

void g (Component* p)
{
  cout << typeid (*p) .name () ;
}
```

Символьное представление имени класса является системнозависимым. Эта С-строка располагается в системной области памяти, так что программист не должен освобождать ее операцией *delete* [].

15.4.4.1. Расширенная информация о типе

Как правило, выявление точного типа объекта является лишь первым шагом получения более детальной информации о типе и его возможностях.

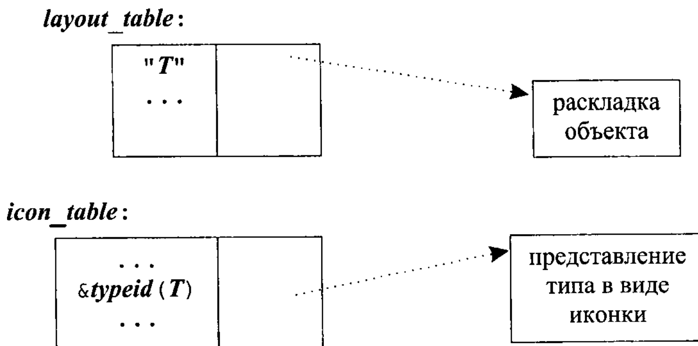
Рассмотрим вопрос о том, как приложение или инструментальное средство может сделать информацию о типе доступной пользователю на этапе выполнения программы. Допустим, что у меня есть инструментальное средство, генерирующее раскладку объектов каждого используемого класса. Я могу поместить эти описания в контейнер типа ***map***, чтобы пользовательский код получил доступ к этой информации:

```
map<string, Layout> layout_table;

void f(B* p)
{
  Layout&x = layout_table[typeid(*p).name()];
  // используем x
}
```

Кто-нибудь еще мог бы предоставить информацию совершенно иного характера:

```
struct TI_eq
{
  bool operator() (const type_info* p, const type_info* q) {return *p==*q;}
};

struct TI_hash
{
  int operator() (const type_info* p);      // вычисляет hash-значение (§17.6.2.2)
};

hash_map<const type_info*, Icon, TI_hash, TI_eq> icon_table;    // §17.6

void g(B* p)
{
  Icon& i = icon_table[&typeid(*p)];
  // используем i
}
```

Такие варианты связывания результатов операции ***typeid*** с информационными структурами позволяют разным людям или инструментам независимо предоставлять разную информацию о типах:



Это очень важно, так как вероятность того, что всех устроит одинаковая информация о типах, близка к нулю.

15.4.5. Корректное и некорректное применение RTTI

Информацию о типах на этапе выполнения следует использовать только в случае реальной необходимости. Статическая (на этапе компиляции) проверка надежнее, влечет меньшие накладные расходы и приводит (как правило) к более структурированным программам. Например, можно использовать RTTI и для написания тонко замаскированных *switch-операторов*:

```
// неразумное применение RTTI:
void rotate (const Shape& r)
{
  if(typeid(r)==typeid(Circle))
  {
    // ничего не делаем
  }
  else if(typeid(r)==typeid(Triangle))
  {
    // вращаем треугольник
  }
  else if(typeid(r)==typeid(Square))
  {
    // вращаем квадрат
  }
  //...
}
```

Применение ***dynamic_cast*** вместо ***typeid*** значительно улучшит этот код.

К сожалению, это не надуманный пример — такой код действительно пишется. Для многих людей, использовавших языки C, Pascal, Modula и Ada, очень трудно отказаться от написания кода в виде *switch-операторов*. Этому следует сопротивляться. В большинстве случаев, когда в зависимости от типа требуется осуществить вариативное поведение кода, вместо RTTI лучше использовать виртуальные функции (§2.5.5, §12.2.6).

Примеры надлежащего применения RTTI возникают, когда сервисный код выражен в терминах некоторого класса, а пользователь хочет добавить функциональность посредством наследования. Хорошей иллюстрацией сказанному может служить семейство классов ***Ival_box*** из §15.4. Если бы пользователь хотел и имел возможность модифицировать библиотечные классы, скажем ***BBwindow***, необходимость в RTTI отпала бы. В противном случае этот механизм нужен. Даже если пользователь хочет модифицировать базовые классы, в такой модификации имеются свои проблемы. Например, возможно пришлось бы писать фиктивные реализации виртуальных функций в классах, для которых они не нужны (или не имеют смысла). Эта проблема обсуждается более подробно в §24.4.3. Применение RTTI для реализации простой системы объектного ввода/вывода рассматривается в §25.4.1.

Люди, имеющие большой опыт работы с языками Smalltalk или Lisp, сильно опираются на динамическую проверку типов и испытывают искушение использовать RTTI совместно с чрезмерно общими типами. Рассмотрим пример:

```
// неразумное применение RTTI:
class Object { /* ... */ };                // полиморфный

class Container: public Object
{
public:
  void put(Object*);
  Object* get();
  // ...
};

class Ship: public Object { /* ... */ };

Ship* f(Ship* ps, Container* c)
{
  c->put(ps);
  // ...
  Object* p = c->get();
  if(Ship* q = dynamic_cast<Ship*>(p))      // проверка на этапе выполнения
  {
    return q;
  }
  else
  {
    // что-нибудь еще (как правило, обработка ошибки)
  }
}
```

Здесь класс *Object* — необязательный артефакт реализации. Он слишком общий и не коррелирует с абстракциями какой-либо предметной области; он заставляет прикладного программиста использовать абстракции реализации. Проблемы такого рода часто решаются применением шаблонных контейнеров, хранящих единственный тип указателей:

```
Ship* f(Ship* ps, list<Ship*>& c)
{
  c.push_front(ps);
  // ...
  return c.pop_front();
}
```

В комбинации с виртуальными функциями этот подход годится для большинства случаев.

15.5. Указатели на члены классов

Многие классы объявляют простые и весьма общие интерфейсы, которые предполагается использовать самыми разными способами. Например. для большинства «объектно-ориентированных» графических интерфейсов пользователя определяется некоторый набор запросов, реакцию на которые должен обеспечить каждый представленный на экране объект. Кроме того, программы могут формировать такие запросы прямо или косвенно. Рассмотрим простой вариант этой идеи:

```
class Std_interface
{
public:
  virtual void start() = 0;
  virtual void suspend() = 0;
  virtual void resume() = 0;
  virtual void quit() = 0;
  virtual void full_size() = 0;
  virtual void small() = 0;

  virtual ~Std_interface() { }
};
```

Точный смысл каждой операции определяется объектом, для которого она вызвана. Часто, между порождающей запрос программой (от имени пользователя) и получающим запрос объектом находится некоторый промежуточный слой кода. В идеале, этот промежуточный код ничего не должен знать про индивидуальные операции типа *resume*() или *full_size*(). В противном случае, промежуточный код пришлось бы модифицировать при каждом изменении набора операций. Следовательно, промежуточный код должен лишь передавать от источника запроса к его получателю некоторые данные, идентифицирующие требуемую операцию.

Одним из простых способов реализации такого подхода является пересылка символьной строки, содержащей название операции. Например, для вызова операции *suspend*() пересылается строка "*suspend*". Естественно, что должен быть кто-то, кто формирует такую строку, и кто-то другой, кто ее декодирует, чтобы выявить необходимую операцию (или отсутствие таковой). Часто такой подход может показаться слишком непрямым и слишком утомительным. Вместо этого можно было бы просто пересылать целые числа, представляющие операции. Например, число *2* могло бы означать *suspend*(). Однако пусть компьютерам и удобно работать с числами, человеку они мало что говорят. Кроме того, все равно нужно писать код, выявляющий, что число *2* соответствует операции *suspend*(), после чего вызывать эту операцию.

Теперь рассмотрим такое средство языка C++, как косвенная ссылка на член класса. Значение указателя на член класса идентифицирует этот член класса. Вы можете думать о нем как о позиции члена класса в объекте этого класса, но компилятор, конечно же, учитывает и различия в членах класса: поля данных класса, виртуальные и невиртуальные функции и т.д.

Рассмотрим интерфейс *Std_interface*. Если я хочу вызвать функцию *suspend*() для некоторого объекта, не указывая эту операцию напрямую, мне потребуется указатель на *Std_interface::suspend*(). Естественно, что мне также будет нужен указатель или ссылка на объект, для которого и нужно вызвать функцию *suspend*(). Рассмотрим тривиальный пример:

```
typedef void (Std_interface::*Pstd_mem)(); // указатель на функцию-член

void f(Std_interface* p)
{
  Pstd_mem s = &Std_interface::suspend;

  p->suspend();                  // прямой вызов
  (p->*s)();                     // вызов по указателю на функцию-член
}
```

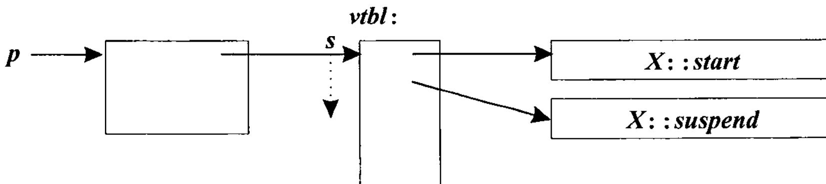
Указатель на член класса (*pointer to member*) можно получить с помощью операции взятия адреса, примененную к полностью квалифицированному имени члена класса, например, &***Std_interface*****::*****suspend***. Переменная типа «указатель на член класса *X*» объявляется с применением декларатора вида *X*::*.

Часто применяют оператор ***typedef*** для улучшения читаемости неудобоваримого синтаксиса деклараторов языка С. Обратите внимание на то, что синтаксис декларатора *X*::* в точности соответствует обычному декларатору * для указателей.

Указатель ***m*** на некоторый член класса можно применять как с указателем на объект, так и с самим объектом, для чего нужно использовать операции ->* и .*, соответственно. Например, выражение ***p->*m*** связывает ***m*** с объектом, на который указывает ***p***, а выражение ***obj.*m*** — связывает m с объектом ***obj***. Результат этих операций используется в соответствии с типом ***m***. Невозможно сохранить результат операций ->* и .* для его дальнейшего использования.

Конечно же, если бы мы заранее знали, какой именно член класса нам нужен, то мы могли бы работать с ним напрямую, а не затевать всю эту катавасию с указателями на члены классов. Как и в случае обычных указателей на функции, мы применяем указатели на функции-члены классов тогда, когда нам нужно сослаться на функцию-член, имя которой неизвестно. В отличие, однако, от обычных указателей на переменные или функции, которые представляют собой готовые целевые адреса в памяти, указатели на члены классов скорее являются смещениями в рамках некоторых структур (или индексами массивов). Когда указатель на член класса комбинируется с указателем на объект класса соответствующего типа, тогда-то и вырабатывается точный адрес конкретного члена для этого конкретного объекта.

Графически это можно изобразить следующим образом:



Так как указатель на виртуальную функцию-член класса (s в нашем примере) является в некотором смысле смещением, то он не зависит от точного расположения объекта в памяти. Поэтому указатель на виртуальную функцию-член можно безопасно передавать из одного адресного пространства в другое адресное пространство, при условии одинаковой раскладки объекта в них обоих. В то же время, указатели на невиртуальные функции-члены (как и обычные указатели на функции) нельзя передавать в другие адресные пространства.

Очевидно, что функции, вызываемые через указатели на функции-члены, могут быть виртуальными. Например, когда мы вызываем ***suspend***() через указатель на функцию-член, вызывается правильная версия функции, соответствующая объекту, к которому применялся указатель на функцию-член. Это является важным аспектом поведения указателей на функции-члены классов.

Интерпретирующий код может использовать указатели на функции-члены для вызова функций-членов, представленных в строковом виде:

```
map<string,Std_interface*> variable;
map<string,Pstd_mem> operation;

void call_member(string var, string oper)
{
  (variable[var]->*operation[oper])();      // var.oper()
}
```

Чрезвычайно полезное применение указателей на функции-члены рассматривается в связи со стандартным шаблоном *mem_fun*() в §3.8.5 и §18.4.

Так как статический член класса не ассоциируется с конкретным объектом класса, то указатель на статический член похож на обычный указатель. Например:

```
class Task
{
  // ...
  static void schedule();
};

void (*p)() = &Task::schedule;            // ok
void (Task::*pm)() = &Task::schedule;     // error: обычный указатель присваивается
                                          // указателю на функцию-член класса
```

Указатели на классовые поля данных рассматриваются в §C.12.

15.5.1. Базовые и производные классы

Производный класс в любом случае содержит члены, достающиеся ему от базовых классов. Кроме того, часто у него есть и собственные члены. Из этого следует, что мы можем безопасно присваивать значения указателей на члены базовых классов указателям на члены производных классов, но не наоборот. Например:

```
class text: public Std_interface
{
public:
  void start();
  void suspend();
  // ...
  virtual void print();

private:
  vector s;
};

void (Std_interface::*pmi)() = &text::print;   // error
void (text::*pmt)() = &Std_interface::start;   // ok
```

Это правило кажется противоположным известному правилу, что можно присваивать значения указателей на производные классы указателям на базовые классы. Однако оба правила действуют абсолютно согласованно: они гарантируют, что указатели никогда не адресуют объекты, у которых отсутствуют свойства, подразумеваемые типом указателя. В нашем примере указатели типа *Std_interface*::* могут использоваться с любыми объектами иерархии *Std_interface*, часть из которых не относится к типу *text*. Следовательно, у них может не быть функции-члена *text*::*print*(),

которым мы пытались проинициализировать ***pmi***. Отказывая в инициализации, компилятор предотвращает ошибки времени выполнения.

15.6. Свободная память

Можно управлять выделением памяти для класса, определив в нем функции ***operator new***() и ***operator delete***() (§6.2.6.2). Однако замена глобальных функций ***operator new***() и ***operator delete***() — занятие не для слабонервных. В конце концов, другие программисты могут рассчитывать на предопределенное поведение механизма работы с динамической памятью, или представить свои собственные версии.

Более ограниченный и более надежный подход состоит в реализации этих операций в рамках конкретного класса. Причем этот класс может быть и базовым для многих других классов. Например, для класса ***Employee*** из §12.2.6 можно было бы определить специализированные операции выделения и освобождения памяти:

```
class Employee
{
  // ...
public:
  // ...
  void* operator new (size_t) ;
  void operator delete (void* , size_t) ;
};
```

Функции-члены operator ***new***() и ***operator delete***() неявно статические, так что у них отсутствует возможность работы с указателем ***this*** и они не изменяют объекты. Они просто предоставляют память, которую конструктор может инициализировать, а деструктор — очищать (деинициализировать):

```
void* Employee: :operator new (size_t s)
{
  // выделить s байт памяти и вернуть указатель на эту память
}

void Employee: :operator delete (void* p, size_t s)
{
  if(p)
  {
    // удаляем только если p!=0; see §6.2.6, §6.2.6.2
    // полагаем, что p указывает на s байт памяти, выделенной при помощи
    // Employee::operator new() и освобождаем эту память для повторного использования
  }
}
```

Использование загадочного до сих пор аргумента типа ***size_t*** теперь становится понятным — это размер фактически уничтожаемого объекта. Если, например, удаляется объект типа ***Employee***, то этот аргумент равен ***sizeof***(***Employee***), а если удаляется объект типа ***Manager*** — то ***sizeof***(***Manager***). Это позволяет специфическому для класса аллокатору не хранить объем выделенной памяти для каждого объекта. Но, естественно, он может и хранить эту информацию (аллокаторы общего назначения

обязаны это делать) и игнорировать аргумент типа ***size_t*** у функции ***operator delete***(). Последний подход затрудняет повышение производительности (в плане скорости и непроизводительных расходов памяти) механизмов работы с памятью по сравнению с таковыми же для общего назначения.

Как компилятор узнает правильное значение второго аргумента (размер удаляемого объекта) при вызове функции ***operator delete*** () ? Пока операция ***delete*** ассоциируется с истинным типом объекта, это просто. Однако так бывает далеко не всегда:

```
class Manager: public Employee
{
  int level;
  // ...
};

void f()
{
  Employee* p = new Manager;      // беда: потерян истинный тип
  delete p;
}
```

В этом случае компилятор не знает истинного размера. Так же, как и при удалении массива, требуется помощь от программиста. Это делается добавлением виртуального деструктора к базовому классу ***Employee***:

```
class Employee
{
public:
  void* operator new(size_t);
  void operator delete(void*, size_t);
  virtual ~Employee();
  // ...
};
```

Подойдет даже пустой деструктор:

```
Employee::~Employee() { }
```

В случае виртуальных деструкторов необходимый для освобождения размер памяти так или иначе связывается с вызовом правильного деструктора (который знает истинный размер объекта).

Присутствие в классе ***Employee*** виртуального деструктора гарантирует, что каждый производный класс будет располагать деструктором (обеспечивающим информацию о правильном размере объекта), даже если таковой в этом производном классе явно и не определяется. Например:

```
void f()
{
  Employee* p= new Manager;
  Delete p;       // теперь все правильно (Employee - полиморфный)
}
```

Выделение кода осуществляется при помощи генерируемого компилятором вызова

```
Employee::operator new(sizeof(Manager))
```

а освобождение памяти — при помощи другого генерируемого компилятором вызова:

```
Employee::operator delete(p,sizeof(Manager))
```

Другими словами, если вы хотите предоставить собственную пару аллокатор/деаллокатор, которая корректно работает с производными классами, то вы должны либо определить виртуальный деструктор в базовом классе, либо воздержаться от использования аргумента типа *size_t* в деаллокаторе. Естественно, можно было бы спроектировать этот аспект языка таким образом, чтобы избавить программиста от сопутствующих сложностей. Но тогда бы и не было возможности осуществлять оптимизацию работы с памятью, присущую лишь менее безопасным системам.

15.6.1. Выделение памяти под массивы

Функции ***operator new***() и ***operator delete***() позволяют программисту брать на себя управление выделением/освобождением памяти для индивидуальных объектов; функции ***operator new*** [] () и ***operator delete*** [] () играют ту же роль для массивов. Например:

```
class Employee
{
public:
  void* operator new[] (size_t);
  void operator delete[] (void*, size_t);
  // ...
};

void f(int s)
{
  Employee* p = new Employee[s];
  // ...
  delete[] p;
}
```

В этом случае память выделяется при помощи вызова

```
Employee::operator new[] (sizeof(Employee)*s+delta)
```

(где ***delta*** — некоторая вспомогательная память, зависящая от реализации), а освобождается память при помощи вызова

```
Employee::operator delete[] (p);    // освобождаем s*sizeof(Employee)+delta байт
```

Количество элементов s и избыточная память ***delta*** запоминаются системой. Если бы в классе ***Employee*** была объявлена двухаргументная версия функции ***operator delete*** [] (), то она бы вызывалась со вторым аргументом, равным *s*sizeof*(*Employee*) + *delta*.

15.6.2. «Виртуальные конструкторы»

После знакомства с виртуальными деструкторами возникает очевидный вопрос: «Может ли конструктор быть виртуальным?». Краткий ответ — нет; чуть менее краткий ответ такой: «Нет, но обеспечить искомый эффект возможно».

Для инициализации (создания) объекта конструктор должен знать его точный тип. Отсюда следует, что конструктор не может быть виртуальным. Более того, конструктор вообще не является обычной функцией. В частности, он взаимодействует со средствами управления памятью не так, как это делают обычные функции. Следовательно, вы не можете располагать указателем на конструктор.

Оба указанных ограничения можно обойти, определив функцию, которая вызывает конструктор и возвращает объект. Это полезная возможность, поскольку часто возникает необходимость в создании объекта, точный тип которого неизвестен. Класс ***Ival_box_maker*** (§12.4.4) служит примером класса, специально спроектированного для решения этой задачи. Теперь же я представлю иную вариацию данной идеи, когда объекты класса могут предоставлять пользователю клоны (копии) самих себя или полностью новые объекты своего собственного типа. Рассмотрим следующий код:

```
class Expr
{
public:
  Expr() ;                  // умолчательный конструктор
  Expr(const Expr&) ;       // копирующий конструктор

  virtual Expr* new_expr() const {return new Expr() ; }
  virtual Expr* clone() const {return new Expr(*this) ; }
  // ...
};
```

Поскольку функции вроде ***new_expr()*** и ***clone()*** являются виртуальными и они (косвенно) конструируют объекты, их часто называют «виртуальными конструкторами» (шутка английского языка). Каждая из этих функций просто вызывает конструктор для создания необходимого объекта.

Производный класс может заместить функции ***new_expr()*** и/или ***clone()***, чтобы они возвращали объекты производного класса:

```
class Cond: public Expr
{
public:
  Cond() ;
  Cond(const Cond&) ;

  Cond* new_expr() const {return new Cond() ; }
  Cond* clone() const {return new Cond(*this) ; }
  // ...
};
```

Это означает, что получив указатель типа ***Expr**** на объект данной иерархии, можно создать объект «точно такого же типа». Например:

```
void user(Expr* p)
{
  Expr* p2 = p->new_expr() ;
  // ...
}
```

Указатель, который здесь присваивается указателю ***p2***, имеет правильный, но неизвестный тип.

Возвраты функций ***Cond::new_expr()*** и ***Cond::clone()*** имеют тип ***Cond****, а не ***Expr****. Это позволяет клонировать ***Cond*** без потери информации о типе. Например:

```
void user2 (Cond* pc, Expr* pe)
{
  Cond* p2 = pc->clone () ;
  Cond* p3 = pe->clone () ;    // error
  // ...
}
```

Тип замещающей функции должен быть в точности таким же, как у замещаемой виртуальной функции, за исключением небольшого послабления по отношению к типу возвращаемого значения. Например, если у исходной функции возврат имел тип ***B****, то возврат у замещающей функции может быть ***D****, где ***B*** является открытым базовым классом для ***D***. Аналогично, вместо ***B***&, тип возврата может быть ослаблен до ***D***&.

Заметьте, что аналогичные правила ослабления для типов аргументов привели бы к нарушению защиты типов (см. §15.8[12]).

15.7. Советы

1. Используйте обычное множественное наследование для того, чтобы выразить объединение свойств; §15.2, §15.2.5.
2. Используйте множественное наследования для отделения деталей реализации от интерфейса; §15.2.5.
3. Используйте *виртуальные* базовые классы там, где требуется выразить общность некоторой части (но не всех) классов иерархии; §15.2.5.
4. Избегайте явных приведений типов; §15.4.5.
5. Используйте ***dynamic_cast*** там, где навигация по иерархии классов неизбежна; §15.4.1.
6. Предпочитайте ***dynamic_cast*** операции ***typeid***; §15.4.4.
7. Предпочитайте доступ ***private*** доступу ***protected***; §15.3.1.1.
8. Не объявляйте поля данных защищенными (***protected***); §15.3.1.1.
9. Если класс определяет функцию ***operator delete()***, он должен иметь виртуальный деструктор; §15.6.
10. Не вызывайте виртуальные функции в процессе создания или уничтожения объекта; §15.4.3.
11. Умеренно используйте явную квалификацию для разрешения перегрузки имен членов класса и применяйте ее в основном в замещающих функциях; §15.2.1.

15.8. Упражнения

1. (*1) Напишите шаблон ***ptr_cast***, который работает как ***dynamic_cast***, только он вместо возврата *нуля* генерирует исключение ***bad_cast***.
2. (*2) Напишите программу, которая иллюстрирует влияние последовательности вызова конструкторов на состояние объекта (с точки зрения RTTI). Аналогичным образом проиллюстрируйте процесс уничтожения объекта.
3. (*3.5) Реализуйте версию настольной игры Reversi/Othello. Каждый игрок может быть либо человеком, либо компьютером. Сфокусируйтесь на корректной работе программы, а затем доведите качество игры компьютера до такого уровня, чтобы с ним было интересно играть.
4. (*3) Улучшите пользовательский интерфейс игры из §15.8[3].
5. (*3) Определите класс графических объектов с достаточным набором операций, чтобы он мог служить в качестве базового класса для графической библиотеки (подсмотрите необходимый набор операций в какой-либо коммерческой графической библиотеке). Определите класс для работы с базами данных, который служил бы базовым классом для библиотеки типов, хранящихся как последовательность полей в базе данных (подсмотрите необходимый набор операций в какой-либо коммерческой системе управления базами данных). Определите графические объекты базы данных, используя подходы со множественным наследованием и без него (сравните преимущества каждого из подходов).
6. (*2) Напишите версию функции ***clone***() из §15.6.2, которая помещала бы клонированные объекты в область памяти типа Arena (см. §10.4.11), переданную в качестве аргумента. Реализуйте простой конкретный класс для работы с фиксированными областями памяти, как производный от абстрактного класса ***Arena***.
7. (*2) Не заглядывая в книгу, напишите как можно больше ключевых слов языка C++.
8. (*2) Напишите удовлетворяющую стандартам программу на C++, содержащую последовательность из по крайней мере десяти разных ключевых слов, не разделенных идентификаторами, знаками операций, пунктуации и т.д.
9. (*2.5) Изобразите возможное распределение памяти для класса ***Radio*** из §15.2.3.1. Объясните, как можно реализовать вызов виртуальной функции.
10. (*2) Изобразите возможное распределение памяти для класса ***Radio*** из §15.2.4. Объясните, как можно реализовать вызов виртуальной функции.
11. (*3) Рассмотрите вопрос о том, как можно реализовать операцию ***dynamic_cast***. Определите и реализуйте шаблон ***dcast***, который ведет себя как ***dynamic_cast***, но использует лишь данные и функции, определенные вами. Проверьте, что вы можете добавлять в систему новые классы, не изменяя определения ***dcast*** и других ранее написанных классов.
12. (*2) Предположим, что правила проверки типов для аргументов ослаблены аналогично правилам для типов возвращаемых значений с тем, чтобы можно было заместить функцию, имеющую аргумент типа ***Base****, на функцию с аргументом ***Derived****. Напишите программу, которая может испортить объект типа ***Derived*** без использования приведения типов. Опишите безопасное ослабление правил для типов аргументов замещаемых функций.

Часть III

Стандартная библиотека

Здесь описывается стандартная библиотека языка C++. Представлены архитектура библиотеки и ключевые методики ее реализации с целью объяснить, как использовать библиотеку эффективным образом. Также на ее примере демонстрируются общие архитектурные идеи и методики программирования. Показывается, как можно расширить библиотеку в соответствии с заложенными для этого возможностями.

Главы

16. Организация библиотеки и контейнеры

17. Стандартные контейнеры

18. Алгоритмы и классы функциональных объектов

19. Итераторы и аллокаторы

20. Строки

21. Потоки

22. Классы для математических вычислений

16

Организация библиотеки и контейнеры

Это было ново. Это было необыкновенно.
Это было просто. Это должно было сработать!
— Г. Нельсон

Критерии проектирования стандартной библиотеки — организация библиотеки — стандартные заголовочные файлы — языковая поддержка — проектирование контейнеров — итераторы — базовые контейнеры — STL-контейнеры — вектор (***vector***) — итераторы — доступ к элементам — конструкторы — модификаторы — операции со списками — размер и емкость — ***vector<bool>*** — советы — упражнения.

16.1. Проектные решения стандартной библиотеки.

Что должно входить в стандартную библиотеку языка С++? Для программиста было бы идеальным найти в библиотеке любые интересные и значимые классы, функции, шаблоны и т.д. Но вопрос задан по отношению именно к стандартной библиотеке, а не по отношению к некоторой библиотеке вообще, так что ответ «Все!» здесь неуместен. Стандартная библиотека должна включать в себя только то, что каждая реализация языка могла бы предоставить надежнейшим образом и на что каждый программист мог бы всегда опереться.

В итоге, стандартная библиотека языка С++:

1. Обеспечивает поддержку таких средств языка, как управление памятью (§6.2.6) и механизм RTTI (§15.4).
2. Предоставляет информацию о зависящих от реализации аспектах языка, таких как, например, максимальное значение типа ***float*** (§22.2).
3. Обеспечивает программиста функциями, которые невозможно одинаково оптимально выполнить лишь средствами самого языка, например ***sqrt***() (§22.3) и ***memmove***() (§19.4.6).

4. Реализует такие непримитивные и переносимые средства, как списки (§17.2.2), ассоциативные массивы (отображения) (§17.4.1), функции сортировки (§18.7.1) и потоки ввода/вывода (глава 21).
5. Формулирует архитектурные принципы, позволяющие расширять базовые средства библиотеки, например, соглашения и средства поддержки, позволяющие программисту обеспечивать ввод/вывод пользовательских типов в стиле ввода/вывода встроенных типов.
6. Служит общим фундаментом для других библиотек.

Кроме того, стандартная библиотека предоставляет некоторые программные средства, вроде генератора случайных чисел (§22.7), просто потому, что так принято делать (и это удобно).

Дизайн стандартной библиотеки в основном определялся последними тремя из перечисленных выше ее ролей. Эти роли, к тому же, связаны между собой. Например, переносимость является важным критерием проектирования специализированных библиотек, а общие контейнерные типы, такие как списки и ассоциативные массивы, важны для обеспечения удобного взаимодействия между независимо разработанными библиотеками.

Последняя роль особо важна с точки зрения дизайна, ибо она помогает очертить область применения стандартной библиотеки и ограничить входящие в нее средства. Например, строки и списки входят в стандартную библиотеку. Если бы их в ней не было, то независимые библиотеки могли бы взаимодействовать лишь только посредством встроенных типов. В то же время, никакой графики или распознавания графических образов в стандартной библиотеке нет. Это все безусловно полезные и широко применимые средства, но они редко когда используются непосредственно при взаимодействии разных библиотек.

В итоге, если некоторое средство не нужно для поддержки перечисленных ролей, его можно не включать в стандартную библиотеку. Это открывает другим библиотекам шанс предоставить конкурентоспособную реализацию такого средства.

16.1.1. Проектные ограничения

Отводимые стандартной библиотеке роли накладывают несколько ограничений на ее дизайн. Средства стандартной библиотеки языка C++ спроектированы так, чтобы они:

1. Были важными и доступными для каждого студента и профессионального программиста, включая разработчиков других библиотек.
2. Использовались прямо или косвенно любым программистом для решения любой задачи, относящейся к области стандартной библиотеки.
3. Были достаточно эффективными, чтобы при реализации иных библиотек эти средства могли составить естественную альтернативу функциям, классам и шаблонам, программируемым вручную.
4. Были независимыми от стратегии реализации (алгоритма) или позволяли пользователю задавать стратегию в качестве аргумента.

5. Были примитивными в математическом смысле, ибо эффективность компонента библиотеки, выполняющего две слабо связанные роли, наверняка будет хуже эффективности компонентов, каждый из которых выполняет лишь одну из ролей.
6. Были удобными, эффективными и безопасными для применения в типичных случаях.
7. Были полностью реализованными. Другим библиотекам можно оставить массу функциональности, но то, что реализовано в стандартной библиотеке, не должно нуждаться в переделках и дополнительной реализации базовых возможностей.
8. Могли сочетаться со встроенными типами и операциями.
9. Были по умолчанию безопасными в отношении типов.
10. Поддерживали общепринятые стили программирования.
11. Были гибкими и расширяемыми и могли работать с пользовательскими типами так же, как со встроенными типами и типами из стандартной библиотеки.

Например, жесткое кодирование критерия сравнения внутри функции сортировки неприемлемо, так как сортировка может производиться согласно разным критериям. Вот почему функция сортировки *qsort*() из стандартной библиотеки языка C принимает функцию сравнения в качестве аргумента, а не полагается на что-то фиксированное вроде операции < (§7.7). С другой стороны, излишние затраты на вызов функции при каждом акте сравнения компрометируют *qsort*() с точки зрения ее роли готового блока для построения других библиотек. Почти для любых типов данных легко выполнить сравнение без лишних затрат на вызов функции.

Велики ли эти затраты? В большинстве случаев — нет. Но в то же время, для некоторых алгоритмов затраты на вызов функции могут составить львиную долю от полного времени работы, что заставит пользователя искать альтернативы. Методика предоставления алгоритма сравнения посредством параметра шаблона, описанная в §13.4, решает проблему. Данный пример показывает конкуренцию между эффективностью и универсальностью. От стандартной библиотеки требуется не просто выполнять поставленные перед ней задачи, но делать это достаточно эффективно, чтобы не провоцировать пользователя на реализацию собственных решений. В противном случае, разработчики более продвинутых и специализированных средств просто вынуждены будут обходить стандартную библиотеку стороной, чтобы оставаться конкурентоспособными. Все это только усложнило бы жизнь и разработчикам библиотек, и их пользователям, желающим оставаться независимыми от конкретной платформы в случае применения нескольких независимых библиотек.

Требования «примитивности» и удобства типового использования кажутся взаимоисключающими. Первое требование мешает особой оптимизации стандартной библиотеки для общих случаев. Однако компоненты для решения общих (часто встречающихся) и непримитивных задач могут быть добавлены к стандартной библиотеке помимо, а не вместо примитивных средств. Культ ортогональности не должен помешать нам сделать жизнь новичка или случайного пользователя стандартной библиотеки простой и удобной. Он не должен заставлять нас мириться с неясным и опасным умолчательным поведением компонентов библиотеки.

16.1.2. Организация стандартной библиотеки

Средства стандартной библиотеки определены в пространстве имен *std* и расположены в некотором наборе заголовочных файлов, реализующих большую часть этих средств. Перечисление этих заголовочных файлов дает представление о стандартной библиотеке и поясняет направление ее рассмотрения в настоящей и последующих главах книги.

Ниже в данном разделе мы приводим список заголовочных файлов стандартной библиотеки, сгруппированный по функциональности, и сопровождаемый краткими пояснениями и ссылками на разделы книги, в которых они рассматриваются.

Стандартный заголовочный файл, начинающийся на букву с, эквивалентен соответствующему заголовочному файлу стандартной библиотеки языка С. Для каждого файла <*X.h*>, определяющего часть стандартной библиотеки языка С в глобальном пространстве имен и в пространстве имен *std*, имеется заголовочный файл <*cX*>, определяющий те же имена исключительно в пространстве имен *std* (см. §9.2.2).

Контейнеры		
<*vector*>	одномерный массив элементов *T*	§16.3
<*list*>	двусвязный список элементов *T*	§17 2.2
<*deque*>	двусторонняя очередь элементов *T*	§ 17.2.3
<*queue*>	очередь элементов *T*	§17.3.2
<*stack*>	стек элементов *T*	§17.3.1
<*map*>	ассоциативный массив элементов *T*	§17.4.1
<*set*>	множество элементов *T*	§17.4.3
<*bitset*>	множество булевских переменных	§17.5.3

Ассоциативные контейнеры *multimap* и *multiset* находятся в файлах <*map*> и <*set*>, соответственно. Контейнер *priority_queue* объявляется в <*queue*>.

Общие средства		
<*utility*>	операции и пары *pairs*	§ 17.1.4, §17.4.1.2
<*functional*>	объекты-функции	§ 18.4
<*memory*>	аллокаторы для контейнеров	§19.4.4
<*ctime*>	время и дата в стиле С	§s.20.5

Заголовочный файл <*memory*> также содержит шаблон *auto_ptr*, призванный сгладить взаимодействие указателей и исключений (§14.4.2).

Итераторы		
<*iterator*>	итераторы и поддержка итераторов	Глава 19

Итераторы позволяют реализовать обобщенные алгоритмы, работающие поверх любых стандартных контейнеров и аналогичных объектов (§2.7.2, §19.2.1).

Алгоритмы		
<algorithm>	общие алгоритмы	Глава 18
<cstdlib>	***bsearch*** () ***qsort*** ()	§18.1.1

Типичный обобщенный алгоритм может применяться к любому интервалу (последовательности) (§3.8, §18.3) контейнерных элементов любого типа. Функции ***bsearch*** () и ***qsort*** () из стандартной библиотеки языка С применимы ко встроенным массивам элементов типов, не имеющих копирующих конструкторов и определяемых пользователем деструкторов (§7.7).

Диагностика		
<exception>	класс исключений	§14.10
<stdexcept>	стандартные исключения	§14.10
<cassert>	макросы ***assert***	§24.3.7.2
<cerrno>	обработка ошибок в стиле С	§20.4.1

Проверка диагностических утверждений (условий) с генерацией исключений описана в §24.3.7.1.

Строки		
<string>	строка элементов ***T***	Глава 20
<cctype>	классификация символов	§20.4.2
<cwctype>	классификация символов из расширенного набора	§ 20.4.2
<cstring>	функции над строками в стиле С	§20.4.1
<cwcharp>	функции над строками «широких» символов в стиле С	§20.4
<cstdlib>	функции над строками в стиле С	§20.4.1

Заголовочный файл ***<cstring>*** объявляет семейство функций ***strlen*** (), ***strcpy*** () и т.д. Заголовочный файл ***<cstdlib>*** объявляет функции ***atof*** () и ***atoi*** (), конвертирующие С-строки в числовые значения.

Ввод/вывод		
<iosfwd>	«опережающие» объявления средств ***ввода/вывода***	§21.1
<iostream>	стандартные объекты ***iostream*** и операции	§21.2.1
<ios>	базовые классы для ***iostream***	§21.2.1
<streambuf>	буферы потоков	§21.6

Ввод/вывод		
<istream>	шаблон потока ввода	§21.3.1
<ostream>	шаблон потока вывода	§21.3.1
<iomanip>	манипуляторы	§21.4.6.2
<sstream>	потоки ввода/вывода в строки из строки	§21.5.3
<cctype>	функции для работы с символами	§20.4.2
<fstream>	потоки ввода/вывода в файлы	§21.5.1
<cstdio>	семейство функций *printf*()	§21.8
<cwchar>	ввод/вывод символов из расширенного набора в стиле *printf*()	§21.8

Манипуляторы — это объекты, предназначенные для управления (манипулирования) состоянием потока (например, для изменения формата вывода чисел с плавающей запятой) (§21.4.6).

Локализация		
<locale>	представляет локальные особенности	§21.7
<clocale>	представляет локальные особенности в стиле C	§21.7

Заголовочный файл *<locale>* посвящен локализации отличий в формате дат, символах обозначения валют и критериях сравнения строк, свойственных разным естественным языкам и культурам.

Поддержка языка		
<limits>	числовые пределы	§22.2
<climits>	макросы числовых скалярных пределов в стиле C	§22.2.1
<cfloat>	макросы пределов чисел с плавающей точкой в стиле C	§22.2.1
<new>	динамическое распределение памяти	§16.1.3
<typeinfo>	поддержка идентификации типов на этапе выполнения	§15.4.1
<exception>	поддержка обработки исключений	§14.10
<cstddef>	языковая поддержка библиотеки C	§6.2.1
<cstdarg>	поддержка функций с переменым числом аргументов	§7.6
<csetjmp>	раскрутка стека в стиле C	§s.18.7
<cstdlib>	завершение программ	§9.4.1.1
<ctime>	системные часы	§D.4.4.1
<csignal>	обработка сигналов в стиле C	§s.18.7

Заголовочный файл <*cstddef*> определяет: тип значений, возвращаемых ***sizeof***( ), ***size_t***, тип результата вычитания указателей и индексов массива, ***ptrdiff_t*** (§6.2.1) и пользующийся дурной репутацией макрос *NULL* (§5.1.1).

Раскрутка стека в стиле языка С (с использованием ***setjmp*** и ***longjmp*** из <***csetjmp***>) не совместима с обработкой исключений (§8.3, глава 14, приложение Е) и ее лучше избегать. В нашей книге раскрутка стека в стиле языка С и сигналы не рассматриваются, так что ссылка относится к стандарту ISO C++.

Numerics		
<***complex***>	комплексные числа и операции с ними	§22.5
<***valarray***>	векторы чисел и операции с ними	§22.4
<***numeric***>	обобщенные числовые операции	§22.6
<***cmath***>	стандартные математические функции	§22.3
<***cstdlib***>	случайные числа в стиле С	§22.7

По историческим причинам ***abs***( ) и ***div***( ) находятся в <***cstdlib***>, а не вместе с остальными математическими функциями в <***cmath***>.

Пользователь и разработчик конкретной реализации стандартной библиотеки не должны добавлять или изымать объявления из стандартных заголовочных файлов. Также не допустимо изменять содержимое этих файлов посредством макросов и локальных определений (§9.2.3). Любая подобная программа или реализация нарушают требования стандарта, а код использующих эти трюки программ непереносим. Даже если такие программы и работают сегодня, следующие версии реализации стандартной библиотеки могут нарушить их работу. Не следуйте по этому пути.

Чтобы применить какое-либо средство стандартной библиотеки, нужно включить соответствующий заголовочный файл. Самостоятельное написание необходимых объявлений *не* соответствует стандарту. Дело в том, что некоторые компиляторы оптимизируют код, отталкиваясь от факта включения стандартных заголовочных файлов, в то время как другие предоставляют усовершенствованные версии ее средств в ответ на включение соответствующих заголовочных файлов. В любом случае, пользователи не могут и не должны знать, как именно реагируют конкретные реализации на включение стандартных заголовочных файлов.

В то же время, пользователи могут специализировать вспомогательные шаблоны, такие как ***swap***( ) (§16.3.9), под конкретные нужды нестандартных библиотек и пользовательских типов.

16.1.3. Непосредственная поддержка языка C++

Небольшая часть стандартной библиотеки осуществляет непосредственную поддержку языковых средств, без которой программы не смогут запуститься и работать.

Библиотечные функции поддержки операций new и ***delete*** обсуждаются в §6.2.6, §10.4.11, §14.4.4 и §15.6; они расположены в заголовочном файле <***new***>.

Механизм RTTI языка C++ опирается на класс ***type_info***, описанный в §15.4.4, и представленный в заголовочном файле <***typeinfo***>.

Стандартные классы исключений рассматриваются в §14.10 и представлены в заголовочных файлах <***new***>, <***typeinfo***>, <***ios***>, <***exception***> и <***stdexcept***>.

Запуск программы и ее завершение рассматриваются в §3.2, §9.4 и §10.4.9.

16.2. Дизайн контейнеров

Контейнер — это объект, содержащий другие объекты. Примерами служат списки, вектора и ассоциативные массивы. В общем случае, объекты можно добавлять в контейнер и удалять их из него.

Естественно, столь общая идея может быть представлена пользователям по-разному. В этом отношении дизайн стандартной библиотеки языка C++ преследует две цели: предоставить максимальную свободу в реализации индивидуальных контейнеров, но обеспечить общий согласованный интерфейс к ним для пользователей. Это помогает оптимизировать код контейнеров при том, что пользовательский код может не зависеть от того, какой конкретный контейнер используется.

Традиционно дизайн контейнеров направлен на то, чтобы отвечать либо первому, либо второму требованию. Контейнерная и алгоритмическая части стандартной библиотеки (часто называемые STL) одновременно обеспечивают и эффективность, и общность. Последующие разделы покажут сильные и слабые стороны традиционного дизайна контейнеров, что поможет лучше понять выбранное для стандартной библиотеки решение.

16.2.1. Специализированные контейнеры и итераторы

Традиционным подходом к реализации векторов и списков является их дизайн в соответствии с предполагаемым использованием:

```
template<class T> class Vector        // оптимальный
{
public:
  explicit Vector(size_t n) ;         // инициализация n объектами со значением T()

  T& operator[] (size_t) ;            // индексация
  //...
};

template<class T> class List          // оптимальный
{
public:
  class Link { /* ... */ };

  List() ;                            // первоначально пустой
  void put(T*) ;                      // поместить перед текущим элементом
  T* get() ;                          // получить текущий элемент
  //...
};
```

Каждый класс определяет операции, оптимальные для них самих, а в реализации операций для каждого класса мы можем выбрать любое подходящее представление, не беспокоясь об иных типах контейнеров. Это позволяет выполнить наиболее эффективную реализацию этих операций. К тому же, наиболее употребительные для

списков и векторов операции, такие как **put**() и **operator**[] () соответственно, невелики по объему и легко допускают встраивание кода.

Типичная работа с контейнерами любых видов состоит в проходе по всем их элементам (просмотр элементов одного за другим). Обычно это делается путем определения класса итератора, подходящего для данного типа контейнера (см. §11.5 и §11.14[7]).

Однако пользователю, осуществляющему итерацию по элементам, часто нет дела до того, в каком именно контейнере они содержатся — в *списке* или *векторе*. В этом случае итерирующий код не должен зависеть от типа контейнера. Желательно, чтобы один и тот же фрагмент пользовательского кода мог работать с обоими типами контейнеров.

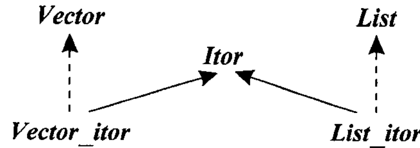
Решение здесь следующее: определить абстрактный класс итератора с операцией «получить следующий элемент», от которого и наследовать для конкретных типов контейнеров. Например:

```
template<class T> class Itor     // общий интерфейс (абстрактный класс §2.5.4, §12.3)
{
public:
// return 0 для индикации состояния «элементов больше нет"

  virtual T* first() = 0;        // указатель на первый элемент
  virtual T* next() = 0;         // указатель на следующий элемент
};
```

Теперь можно предоставить реализации для *векторов* и *списков*:

```
template<class T> class Vector_itor: public Itor<T>  // реализация Vector
{
  Vector<T>& v;
  Size_t index;                                      // индекс текущего элемента

public:
  Vector_itor(Vector<T>& vv) : v(vv), index(0) { }
  T* first() { return (v.size())?&v[index=0]:0; }
  T* next() { return (++index<v.size())?&v[index]:0; }
};

template<class T> class List_itor: public Itor<T>    // реализация List
{
  List<T>& lst;
  List<T>::Link p;                                   // указывает на текущий элемент

public:
  List_itor(List<T>&);
  T* first();
  T* next();
};
```

Отобразим это решение графически, применяя пунктирные линии для отображения отношения «реализовано с использованием»:



Внутреннее устройство двух итераторов совершенно разное, но пользователей это не касается. Можно итерировать что угодно, для чего имеется реализация ***Itor***. Например:

```
int count(Itor<char>& ii, char term)
{
  int c = 0;
  for(char* p = ii.first(); p; p=ii.next()) if(*p==term) c++;
  return c;
}
```

Здесь, однако, имеется одна неприятность. Несмотря на то, что операции с итераторами типа ***Itor*** весьма просты, они все же вносят дополнительные накладные расходы на вызов виртуальных функций. Во многих случаях эти затраты незначительны по сравнению с объемом основной работы. Но в системах, где требуется особая эффективность, итерации поверх простых контейнеров становятся критическими операциями, в которых вызов функции многократно дороже целочисленного сложения или разыменования указателя, выполняемых методом ***next***() для *векторов* и *списков*. Поэтому рассмотренная модель неприменима или плохо применима для стандартной библиотеки.

Однако на практике рассмотренная контейнерно-итераторная модель успешно применяется во многих программных системах. Много лет я сам применял ее для большинства приложений. У нее есть свои достоинства и недостатки:

+ Контейнеры индивидуального дизайна просты и эффективны.

+ От контейнеров не требуется особой общности. Итераторы и классы-обертки (§25.7.1) помогают интегрировать индивидуально спроектированные контейнеры в общую среду разработки.

+ Единообразность применения обеспечивается итераторами (а не общим базовым контейнерным классом; §16.2.2).

+ Для одного и того же контейнера можно определить разные итераторы с разными целями.

+ Контейнеры по умолчанию безопасны по типу элементов и однородны по нему же (все элементы имеют один и тот же тип). Разнородные контейнеры реализуются как однородные контейнеры указателей на общий базовый тип.

+ Контейнеры неинтрузивны (ненавязчивы) — элементы контейнера не обязаны иметь общий базовый класс или специальное связующее поле. Неинтрузивные контейнеры одинаково хорошо работают и со встроенными типами данных, и со ***структурами***, раскладка которых жестко задается извне.

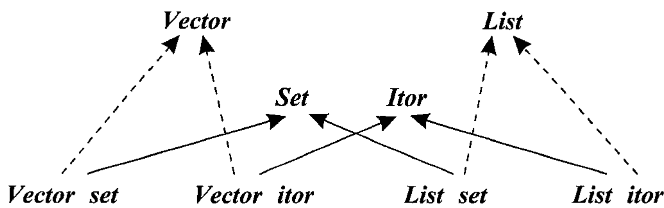
- Каждое обращение к итератору влечет за собой вызов виртуальной функции с временными затратами, значительными по сравнению с простыми встраиваемыми функциями доступа.

- Иерархия классов итераторов по мере развития постепенно запутывается.
- Контейнеры между собой не имеют ничего общего, равно как и содержащиеся в них объекты. Все это препятствует созданию универсальных служб долговременного хранения объектов (persistence services) и объектного ввода/вывода.

Здесь знаком + помечены достоинства, а знаком - помечены недостатки.

Я придаю большое значение гибкости, которую обеспечивают итераторы. Их общий интерфейс, такой как ***Itor***, может быть реализован много позже этапов проектирования и реализации самих контейнеров (здесь это ***Vector*** и ***List***). Начиная проектировать, мы обычно сначала порождаем что-нибудь достаточно конкретное. Например, сначала у нас появляются просто массив и список. И лишь после этого мы обнаруживаем абстракцию, объединяющую массивы и списки в некотором контексте.

Фактически мы можем выполнять такую «позднюю абстракцию» многократно. Предположим, нам потребовалось реализовать абстракцию «множество». Хотя эта абстракция и отличается от абстракции ***Itor***, мы все равно можем реализовать интерфейс ***Set*** (множество) способом, похожим на способ реализации интерфейса ***Itor*** для ***Vector*** и ***List***:



Таким образом, поздняя абстракция посредством абстрактных классов позволяет нам обеспечивать разные реализации концепций, даже если между ними нет сильного сходства. Например, списки и вектора имеют очевидные общие черты, но ведь можно реализовать ***Itor*** и для ***istream***.

В представленном выше списке два последних пункта являются главными недостатками рассматриваемого подхода. То есть даже если для итераторов и иных подобных интерфейсов устранить накладные расходы на вызов функций (что в ряде случаев возможно), то все равно этот подход не станет идеальным для стандартной библиотеки.

Неинтрузивные контейнеры в ряде конкретных случаев требуют большего расхода памяти и времени по сравнению с интрузивными контейнерами, но я не вижу в этом большой проблемы — при необходимости итератор вроде ***Itor*** можно реализовать и для интрузивного контейнера (§16.5[11]).

16.2.2. Контейнеры с общим базовым классом

Интрузивный контейнер можно определять и без шаблонов или иных средств параметризации типа. Например:

```
struct Link
{
  Link* pre;
  Link* suc;
  // ...
};

class List
{
  Link* head;
  Link* curr;           // текущий элемент

public:
  Link* get();          // удалить и вернуть текущий элемент
  void put(Link*);      // вставить перед текущим элементом
  // ...
};
```

Теперь *List* это список структур типа *Link*, и в нем можно хранить объекты любых типов, производных от *Link*. Например:

```
class Ship : public Link { /* ... */ };

void f(List* lst)
{
  while (Link* po = lst->get())
  {
    if (Ship* ps = dynamic_cast<Ship*>(po))     // Ship должен быть полиморфным (§15.4.1)
    {
      // используем Ship
    }
    else
    {
      // Oops, делаем что-нибудь другое
    }
  }
}
```

Язык *Simula* определяет свои стандартные контейнеры подобным образом, так что для языков объектно-ориентированного программирования такой подход можно считать врожденным. В наши дни общий класс всех объектов принято называть *Object* или чем-то в этом роде. Класс *Object* обычно предоставляет и другие полезные услуги помимо обеспечения связи контейнеров.

Часто, но не всегда, этот подход расширяется до предоставления общего контейнерного типа:

```
class Container : public Object
{
public:
  virtual Object* get();                    // удалить и вернуть текущий элемент
  virtual void put(Object*);                // вставить перед текущим элементом
  virtual Object*& operator[](size_t);      // индексация
  // ...
};
```

Операции контейнера ***Container*** виртуальные, так что их можно соответствующим образом замещать для частных случаев контейнеров:

```
class List: public Container
{
public:
  Object* get() ;
  void put(Object*) ;
  // ...
};

class Vector: public Container
{
public:
  Object*& operator[] (size_t) ;
  // ...
};
```

Сразу же обнаруживается одна проблема — какие операции мы хотим получить от класса ***Container***? Мы можем обеспечить здесь только операции, общие для всех контейнеров. Однако пересечение множества операций всех контейнеров — до смешного узкий интерфейс. Более того, во многих практически интересных случаях такое пересечение вообще пусто. Поэтому исходя из реальности мы должны предоставить объединение наиболее существенных операций для множества контейнеров, которое мы намереваемся поддержать. Такое объединение часто называют *жирным интерфейсом* (*fat interface*) (§24.4.3).

В рамках жирного интерфейса мы можем либо обеспечить поведение функций по умолчанию, либо сделать их чисто виртуальными, заставив, тем самым, все производные классы предоставлять собственные реализации этих функций. В любом случае мы получаем множество функций, которые просто сообщают об ошибках времени выполнения. Например:

```
class Container: public Object
{
public:
  struct Bad_op     // класс исключений
  {
    const char* p;
    Bad_op(const char* pp) :p(pp) { }
  };

  virtual void put(Object*) {throw Bad_op("put") ; }
  virtual Object* get() {throw Bad_op("get") ; }
  virtual Object*& operator[] (int) {throw Bad_op("[]") ; }
  // ...
};
```

Для защиты от ситуации, когда контейнер не поддерживает функцию ***get***(), мы просто перехватываем исключение ***Container***::***Bad_op***. Для иллюстрации перепишем пример с классом ***Ship*** следующим образом:

```
class Ship : public Object { /* ... */ };

void f1 (Container* pc)
{
  try
  {
    while (Object* po = pc->get ( ) )
    {
      if (Ship* ps = dynamic_cast<Ship*> (po) )
      {
        // используем Ship
      }
      else
      {
        // Oops, делаем что-нибудь другое
      }
    }
  }
  catch (Container: :Bad_op& bad)
  {
    // Oops, делаем что-нибудь другое
  }
}
```

Все это утомительно, поэтому можно положиться на перехват ***Bad_op*** где-либо в ином месте программы, что заметно упрощает наш пример:

```
void f2 (Container* pc)
{
  while (Object* po = pc->get ( ) )
  {
    Ship& s = dynamic_cast<Ship&> (*po) ;
    // используем Ship
  }
}
```

Однако упование на проверку исключений в иных местах программы я считаю неэффективным, безвкусным, и в итоге предпочитаю в подобного рода случаях статическую проверку:

```
void f3 (Itor<Ship>* i)
{
  while (Ship* ps = i->next ( ) )
  {
    // используем Ship
  }
}
```

У подхода к построению контейнеров на базе общего родительского класса есть свои достоинства и недостатки:

- Операции над конкретными контейнерами требуют дополнительных затрат на вызов виртуальных функций.

- – Все типы контейнеров должны быть производными от ***Container***. Это чревато применением «жирных интерфейсов», требует большой степени предвидения и приводит к необходимости проверки типов на стадии выполнения. Интеграция независимо разработанных контейнеров в общую среду разработки по меньшей мере неудобна (см. §16.5[12]).
- + Общий базовый класс ***Container*** повышает взаимозаменяемость контейнеров, предоставляющих схожие наборы операций.
- – Контейнеры разнородны и в общем случае небезопасны по типу элементов (единственное, на что можно положиться, так это то, что элементы относятся к типу ***Object****). При необходимости безопасные и однородные контейнеры могут определяться шаблонами.
- – Контейнеры интрузивны (каждый элемент должен иметь тип, производный от ***Object***). Объекты встроенных типов и структуры, чья раскладка жестко задается извне, поместить в контейнер непосредственным образом невозможно.
- – Извлеченный из контейнера элемент требует явного приведения типа до момента использования этого элемента.
- + Класс ***Container*** и класс ***Object*** служат базой для решения проблемы реализации универсальных сервисов, пригодных для любых контейнеров и любых объектов, таких как долговременное хранение объектов и объектный ввод/вывод.

Здесь знак плюс маркирует достоинство, а знак минус — недостаток.

По сравнению с подходом, использующим несвязанные контейнеры и итераторы, подход с общим базовым классом всех объектов выплескивает сложности на голову пользователя, вызывает значительные дополнительные расходы времени и памяти и налагает ограничения на объекты, которые можно поместить в контейнер. Кроме того, для многих классов наследование от ***Object*** означает открытие деталей реализации. Таким образом, данный подход далек от того, что требуется для построения стандартной библиотеки.

Однако гибкость и универсальность этого подхода не нужно недооценивать. Его можно с успехом использовать во многих прикладных программах, где не столь важна эффективность, а нужна простота и универсальность, обеспечиваемая единым интерфейсом класса ***Container*** и реализациями универсальных служб, вроде ввода/вывода объектов.

16.2.3. Контейнеры STL

Контейнеры и итераторы стандартной библиотеки (часто называемые STL-ядром этой библиотеки, §3.10) можно считать попыткой объединения лучших черт двух традиционных моделей, рассмотренных выше. Однако на самом деле библиотека STL разрабатывалась иначе — она была построена в результате целенаправленного поиска бескомпромиссно эффективных обобщенных алгоритмов.

Такое стремление к эффективности привело к переходу от плохо поддающихся встраиванию виртуальных функций к небольшим, часто используемым функциям доступа (access functions). Поэтому мы не можем представить стандартный интерфейс к контейнерам или стандартный интерфейс итераторов как абстрактный класс. Вместо этого, каждый тип контейнера поддерживает стандартный набор ба-

зовых операций. Для того чтобы избежать проблемы жирных интерфейсов (§16.2.2, §24.4.3), операции, которые невозможно эффективно реализовать для всех контейнеров, исключаются из стандартного набора общих операций. К примеру, для ***vector*** обращение по индексу введено, а для ***list*** — нет. Дополнительно, каждый вид контейнеров определяет свои собственные итераторы, предоставляющие стандартный набор операций.

Стандартные контейнеры не имеют общего базового класса. Вместо этого каждый контейнер полностью реализует стандартный контейнерный интерфейс. Аналогично, не существует общего базового класса для итераторов. Стандартные контейнеры и итераторы не обеспечивают никакой проверки типов (явной или неявной) во время выполнения.

Важный и сложный вопрос с обеспечением полезных общих сервисов для всех контейнеров решен посредством аллокаторов (allocators), передаваемых в виде шаблонных параметров (а не посредством общего базового класса) (§19.4.3).

У STL-подхода к построению контейнеров есть свои достоинства и недостатки:

+ Каждый конкретный контейнер прост и эффективен (не столь прост, как контейнер индивидуального дизайна, но столь же эффективен).

+ Каждый контейнер обеспечивает набор стандартных операций со стандартными именами и семантикой. Дополнительные операции по мере необходимости реализуются индивидуально отдельными типами контейнеров. Кроме того, классы-обертки (wrapper classes) (§25.7.1) применяются для интеграции индивидуальных контейнеров в общую среду разработки (§16.5[14]).

+ Дополнительная степень универсальности использования контейнеров достигается с помощью итераторов. Каждый контейнер предоставляет итераторы, поддерживающие набор стандартных операций со стандартными именами и семантикой (смыслом). Тип итераторов определяется индивидуально для каждого вида контейнеров так, чтобы итераторы были как можно проще и эффективнее.

+ Кроме стандартных итераторов для удовлетворения разных нужд разных контейнеров могут вводиться различные дополнительные итераторы и обобщенные интерфейсы.

+ По умолчанию контейнеры безопасны в отношении типов и однородны (все элементы контейнеров имеют одинаковый тип). Гетерогенные контейнеры могут создаваться как однородные контейнеры указателей на общий базовый класс.

+ Контейнеры неинтрузивны (от их элементов не требуется иметь специальный родительский класс или связующее поле). Неинтрузивные контейнеры хорошо работают как со встроенными типами, так и со *структурами*, раскладка которых жестко задается извне.

+ Интрузивные контейнеры можно интегрировать в общую среду разработки, но они все равно будут накладывать ограничения на элементы.

+ Каждый контейнер принимает шаблонный аргумент ***allocator***, используемый как средство реализации общих сервисов для всех контейнеров. Это упрощает создание таких сервисов, как долговременное хранение объектов и объектный ввод/вывод (§19.4.3).

- Не существует стандартного представления контейнеров и итераторов для передачи их в функции в качестве аргументов с контролем ошибок времени выполнения (но для конкретных приложений это сделать нетрудно; §19.3).

Как и ранее, + означает достоинство, а знак - означает недостаток.

Итак, контейнеры и итераторы не имеют фиксированного стандартного представления. Вместо этого, каждый контейнер предоставляет стандартный интерфейс в виде стандартного набора операций, так что такие контейнеры взаимозаменяемы. То же справедливо и для итераторов. Все это обеспечивает минимум нагрузки на память и максимум производительности при сохранении общности и на уровне контейнеров (как в случае подхода с общим базовым классом), и на уровне итераторов (как в случае специализированных контейнеров).

STL-подход в огромной степени опирается на шаблоны. Чтобы избежать излишнего реплицирования кода, для реализации контейнеров указателей обычно применяется частичная специализация (§13.5).

16.3. Контейнер типа vector

Мы используем контейнерный тип ***vector*** как пример законченного стандартного контейнера. Все, что явно не оговаривается, относится и ко всем остальным стандартным контейнерам. В главе 17 рассматриваются специальные черты контейнеров ***list***, ***set***, ***map*** и т.д. Средства, предлагаемые контейнером ***vector*** (и другими аналогичными контейнерами), рассматриваются довольно подробно. Цель — понять возможности применения векторов и их роль в общем дизайне стандартной библиотеки.

Обзор стандартных контейнеров и их средств можно найти в §17.1. Ниже мы представляем контейнер ***vector*** поэтапно: типы, итераторы, доступ к элементам, конструкторы, стековые и списковые операции, размер и емкость, вспомогательные функции (helper functions) и специализация ***vector<bool>***.

16.3.1. Типы

Стандартный контейнер ***vector*** является шаблоном, определенным в пространстве имен ***std*** и расположенным в заголовочном файле ***<vector>***. Сначала он определяет ряд стандартных имен типов:

```
template<class T, class A = allocator<T> > class std::vector
{
public:
  // типы:
  typedef T value_type;                          // тип элементов
  typedef A allocator_type;                      // тип менеджера памяти
  typedef typename A::size_type size_type;
  typedef typename A::difference_type difference_type;

  typedef implementation_dependent1 iterator;          // T*
  typedef implementation_dependent2 const_iterator;    // const T*
  typedef std::reverse_iterator<iterator> reverse_iterator;
  typedef std::reverse_iterator<const_iterator> const_reverse_iterator;
```

```
    typedef typename A::pointer pointer;                   // указатель на элемент
    typedef typename A::const_pointer const_pointer;
    typedef typename A::reference reference;               // ссылка на элемент
    typedef typename A::const_reference const_reference;
  //...
  };
```

Каждый стандартный контейнер определяет эти имена типов в качестве своих членов. Каждый определяет их способом, наиболее приемлемым для его реализации.

Тип элементов контейнера передается в виде первого параметра шаблона и известен как ***value_type***. Тип ***allocator_type*** (тип аллокаторов — менеджеров-распределителей памяти), который факультативно передается вторым параметром шаблона, определяет, как ***value_type*** взаимодействует с различными механизмами управления памятью. В частности, аллокатор предоставляет функции, которые контейнер использует для выделения/освобождения памяти под свои элементы. Аллокаторы изучаются в §19.4. Как правило, тип ***size_type*** используется для индексации элементов контейнера, а ***difference_type*** — для типа результата вычитания двух итераторов одного и того же контейнера. В общем случае эти типы соответствуют ***size_t*** и ***ptrdiff_t*** (§6.2.1).

В приложении Е рассматривается, как ведет себя вектор, если распределители памяти (аллокаторы) или операции с элементами генерируют исключения.

С итераторами мы познакомились в §2.7.2, а более подробно они рассматриваются в главе 19. О них можно думать, как об указателях на элементы контейнеров. Каждый контейнер определяет тип ***iterator*** для адресации своих элементов. Тип ***const_iterator*** применяется тогда, когда элементы используются только «для чтения». Как и в случае со встроенными указателями, безопаснее применять *константные* версии итераторов, если нет причин поступать иначе. Фактические типы итераторов зависят от реализации. Их очевидными определениями для традиционных *векторов* являются типы ***T**** и ***const T****, соответственно.

Типы обратных итераторов для *векторов* конструируются из стандартного шаблона ***reverse_iterator*** (§19.2.5). Они представляют последовательность элементов (интервал) в обратном порядке.

Как показано в §3.8.1, знание имен типов, определенных в контейнере, позволяет пользователю писать программы для работы с контейнером, ничего при этом не зная о фактических типах. Особо отметим, что это позволяет написать код, который будет работать с любым стандартным контейнером. Например:

```
template<class C> typename C::value_type sum(const C& c)
{
  typename C::value_type s = 0;
  typename C::const_iterator p = c.begin(); // стартуем с самого начала

  while(p!=c.end())                         // подсчет ведем до самого конца
  {
    s += *p;    // получаем значение элемента
    ++p;        // теперь p указывает на следующий элем-т
  }
  return s;
}
```

Необходимость добавлять ключевое слово ***typename*** перед каждым именем типа из параметра шаблона весьма утомительна. Однако компилятор не экстрасенс, и у него нет иного способа узнать, что имя члена шаблонного параметра является именем типа (§С.13.5).

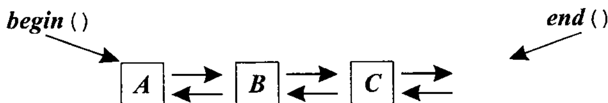
Как и в случае указателей, операция * означает разыменование итератора (§2.7.2, §19.2.1), а операция ++ означает его инкрементирование.

16.3.2. Итераторы

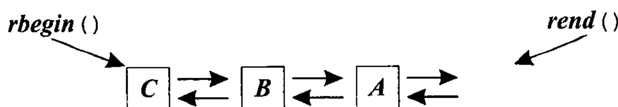
Как было показано в предыдущих разделах, итераторы позволяют пользователю осуществлять навигацию по элементам контейнера без знания применяемых фактических типов. Немногочисленные ключевые функции-члены позволяют пользователю фиксировать концы последовательности элементов:

```
template<class T, class A=allocator<T> > class vector
{
public:
  // ...
  // iterators:

  iterator begin () ;               // указывает на первый элемент
  const_iterator begin () const;

  iterator end () ;                 // указывает на элемент, расположенный за последним
  const_iterator end () const;

  reverse_iterator rbegin () ;      // указывает на первый элем-т обратной последоват-ти
  const_reverse_iterator rbegin () const;

  reverse_iterator rend () ;        // указывает на элемент, расположенный за последним
                                    // элементом обратной последовательности
  const_reverse_iterator rend () const;
  // ...
};
```

Пара функций ***begin***() /***end***() предоставляет доступ к элементам контейнера в обычном порядке. То есть за нулевым элементом следует первый элемент, затем второй элемент и т.д. Пара функций ***rbegin***() /***rend***() предоставляет элементы в обратном порядке. То есть за (***n***–***1***)-ым элементом следует (***n***–***2***)-ой элемент, затем (***n***–***3***)-ий и т.д. Вот как видится последовательность с помощью прямого итератора ***iterator***:



А вот как она представляется обратным итератором ***reverse_iterator*** (§19.2.5):



Это позволяет использовать алгоритмы, которые просматривают элементы в обратном порядке. Например:

```
template<class C> typename C::iterator find_last(C& c, typename C::value_type v)
{
  typename C::reverse_iterator ri = find(c.rbegin(), c.rend(), v);
  if(ri == c.rend()) return c.end(); // используем c.end() для индикации "не найдено"

  typename C::iterator i = ri.base(); // извлекаем итератор из обратного итер-ра (§19.2.5)
  return --i;
}
```

Для ***reverse_iterator***, ***ri.base***() возвращает ***iterator***, указывающий на одну позицию дальше, чем ***ri*** (§19.2.5). Без обратных итераторов пришлось бы явным образом использовать цикл:

```
template<class C> typename C::iterator find_last(C& c, typename C::value_type v)
{
  typename C:: iterator p = c. end ( ); // ищем с конца в начало
  while(p!=c.begin())
    if(*--p==v) return p;
  return c.end();                         // используем c.end() для индикации "не найдено"
}
```

Обратный итератор — это совершенно обычный итератор, так что можно написать и следующий код:

```
template<class C> typename C::iterator find_last(C& c, typename C::value_type v)
{
  typename C::reverse_iterator p = c.rbegin(); // обход последов-ти в обратном порядке

  while(p!=c.rend())
  {
    if(*p==v)
    {
      typename C::iterator i = p.base();
      return --i;
    }
    ++p;               // внимание: инкремент, а не декремент
  }
  return c.end( );     // используем c.end() для индикации "не найдено"
}
```

Стоит подчеркнуть, что ***C::reverse_iterator*** не тот же тип, что ***C::iterator***.

16.3.3. Доступ к элементам

Важным аспектом *векторов* по сравнению с другими контейнерами является возможность простого и эффективного произвольного доступа к его элементам:

```
template<class T, class A = allocator<T> > class vector
{
public:
  // ...
  // доступ к элементам:
```

```
    reference operator[] (size_type n);      // доступ без проверки
    const_reference operator[] (size_type n) const;

    reference at(size_type n);               // доступ с проверкой
    const_reference at(size_type n) const;
    reference front();                       // первый элемент
    const_reference front() const;
    reference back();                        // последний элемент
    const_reference back() const;
    // ...
};
```

Индексирование выполняется как операцией ***operator***[] (), так и именованной функцией-членом ***at***(); при первом варианте доступа контроля выхода за границы индексов не происходит, а во втором при этом генерируется исключение ***out_of_range***. Например:

```
void f(vector<int>& v, int i1, int i2)
try
{
  for(int i=0; i < v.size(); i++)
  {
    // здесь диапазон гарантирован: используем v[i] без проверки
  }

  v.at(i1) = v.at(i2);                  // доступ с проверкой диапазона
  // ...
}
catch(out_of_range)
{
  // oops: ошибка диапазона
}
```

Здесь иллюстрируется идея о том, что если диапазон индексов уже проверен, то можно безопасно использовать операцию индексирования; в противном случае, будет разумно применить функцию ***at***(), осуществляющую соответствующую проверку. Такие различия важно принимать к сведению в случаях, когда важна эффективность. Если же эффективность не столь важна или неизвестно, был ли проверен диапазон индексов, можно применить вектор с операцией индексации, проверяющей диапазоны (см. шаблон ***Vec*** из §3.7.2) или итератор с проверкой (§19.3).

По умолчанию доступ к элементам осуществляется без проверок (как для встроенных массивов). Поверх быстрого средства всегда можно построить безопасное (с проверкой) средство, а вот поверх медленного быстрое построить невозможно.

Операции доступа возвращают значения типа ***reference*** или ***const_reference*** в зависимости от того, применяются ли они к константному или неконстантному объектам. Тип ***reference*** — это некоторый тип, удобный для доступа к элементам. В простой реализации контейнера ***vector<X>*** тип ***reference*** есть просто ***X***&, а ***const_reference*** есть ***const X***&. Результат попытки создания ссылки за пределами допустимого диапазона индексов не определен. Например:

```
void f(vector<double>& v)
{
  double d = v[v.size()];      // не определено: плохой индекс

  list<char> lst;
  char c = lst.front();        // не определено: list - пустой
}
```

Из стандартных последовательных контейнеров только ***vector*** и ***deque*** (§17.2.3) поддерживают индексирование. Это сделано для того, чтобы не смущать пользователей реализацией заведомо неэффективных операций. Например, можно конечно предоставить операцию индексирования списков (§17.2.2), но эта операция весьма неэффективна (порядка ***O***(***n***)).

Функции-члены ***front***() и ***back***() возвращают ссылки на первый элемент и последний элемент, соответственно. Они наиболее полезны в случаях, когда заведомо известно, что эти элементы существуют, и когда эти элементы представляют особый интерес. Очевидным примером является контейнер vector, используемый как стек (§16.3.5). Заметьте, что ***front***() возвращает ссылку на элемент, для которого ***begin***() возвращает итератор. Я часто представляю себе ***front***() как первый элемент, а ***begin***() — как указатель на первый элемент. Соответствие между ***back***() и ***end***() менее очевидно: ***back***() — это последний элемент, а ***end***() указывает на позицию элемента, расположенного в памяти за последним элементом.

16.3.4. Конструкторы

Естественно, что ***vector*** обеспечивает полный набор конструкторов (§11.7), деструктор и операции копирования:

```
template<class T, class A = allocator<T> > class vector
{
public:
  // ...
  // конструкторы и т.п.

  explicit vector(const A& = A());
  explicit vector(size_type n, const T& val=T(), const A&=A());  // n копий val

  template<class In>                          // In - итератор ввода (§19.2.1)
  vector(In first, In last, const A& = A());  // копирование из [first:last[

  vector(const vector& x);

  ~vector();

  vector& operator=(const vector& x);

  template<class In>                     // In - итератор ввода (§19.2.1)
  void assign(In first, In last);        // копирование из [first:last[

  void assign(size_type n, const T& val);  // n копий val
  // ...
};
```

Контейнер ***vector*** обеспечивает быстрый доступ к произвольному элементу, но изменение его размера является дорогостоящей операцией. Поэтому при создании *вектора* ему лучше явно приписать начальный размер. Например:

```
vector<Record> vr(10000);

void f(int s1, int s2)
{
  vector<int> vi(s1);
  vector<double>* p = new vector<double>(s2);
}
```

Элементы векторов, создаваемых таким образом, инициализируются конструкторами по умолчанию (соответствующими типу элементов). То есть каждый из ***10000*** элементов контейнера ***vr*** инициализируется при помощи конструктора ***Record***(), и каждый из ***s1*** элементов контейнера ***vi*** инициализируется с помощью ***int***(). Заметьте, что для встроенных типов выполняется умолчательная инициализация соответствующим ***нулем*** (§4.9.5, §6.2.8).

Если у типа нет конструктора по умолчанию, невозможно создать вектор с элементами этого типа без явного указания значения всех элементов. Например:

```
class Num                          // любая точность
{
public:
  Num(long);
  // нет умолчательного конструктора
  // ...
};

vector<Num> v1(1000);              // error: нет умолчательного конструктора
vector<Num> v2(1000,Num(0));       // ok
```

Поскольку ***вектор*** не может иметь отрицательное число элементов, его размер должен иметь неотрицательное значение. Это отражено в требовании, чтобы для вектора тип ***size_type*** должен быть беззнаковым типом (***unsigned***). Это в некоторых аппаратных архитектурах допускает больший диапазон размеров векторов. Однако это может приводить и к сюрпризам:

```
void f(int i)
{
  vector<char> vc0(-1);   // здесь компилятору легко выдать предупреждение
  vector<char> vc1(i);
}

void g()
{
  f(-1);                  // обманываем f(), передавая ей большое положительное число!
}
```

При вызове ***f***(-***1***) число -***1*** преобразуется в (довольно большое) положительное число (§C.6.7). При удаче компилятор может и предупредить нас об этом.

Размер ***вектора*** можно указать неявно, предоставив начальные значения элементов. Это выполняется передачей конструктору интервала, по которому и нужно создать контейнер типа ***vector***. Например:

```
void f(const list<X>& lst)
{
  vector<X> v1(lst.begin(),lst.end());   // копируем элементы из list
```

```
    char p[] = "despair";
    vector<char> v2(p, &p[sizeof(p)-1]);    // копируем символы из C-строки
}
```

Конструктор класса *vector* подгоняет размер создаваемого вектора под количество элементов из предоставленного интервала.

Конструкторы класса *vector*, принимающие единственный аргумент, объявляются с модификатором *explicit* во избежание предотвращения случайных преобразований (§11.7.1). Например:

```
vector<int> v1(10);                 // ok: вектор из 10 int
vector<in > v2 = vector<in >(10);   // ok: вектор из 10 int
vector<in > v3 = v2;                // ok: v3 - копия v2
vector<in > v4 = 10;  // error: попытка неявного преобразования 10 в vector<int>
```

Копирующий конструктор и перегруженная операция присваивания копируют элементы *вектора*. В случае *векторов* с большим числом элементов такие операции весьма дороги, так что вектора передают в функции чаще всего по ссылке. Например:

```
void f1(vector<int>&);              // обычный стиль
void f2(const vector<int>&);        // обычный стиль
void f3(vector<int>);               // редкий стиль

void h()
{
    vector<int> v(10000);
    //...
    f1(v);          // передаем ссылку
    f2(v);          // передаем ссылку
    f3(v);          // копируем 10000 элементов в новый вектор для f3()
}
```

Функция-член *assign*() предоставляет альтернативу конструкторам со многими аргументами. Она нужна, поскольку операция = принимает единственный правосторонний операнд, так что *assign*() используется тогда, когда требуется значение по умолчанию или диапазон значений. Например:

```
class Book
{
    //...
};

void f(vector<Num>& vn, vector<char>& vc, vector<Book>& vb, list<Book>& lb)
{
    vn.assign(10,Num(0));               // присвоить вектору vn 10 копий Num(0)

    char s[] = "literal";
    vc.assign(s, &s[sizeof(s)-1]);      // присвоить "literal" вектору vc
    vb.assign(lb.begin(), lb.end());    // присваиваем вектору vb элементы списка lb
    //...
}
```

Таким образом, имеются две возможности — инициализировать *вектор* последовательностью элементов того же самого типа или присвоить такую последовательность вызовом *assign*(). Важно, что все это выполняется без введения сонма

конструкторов и функций преобразования. Отметим еще, что присваивание полностью изменяет значения элементов *вектора*. По идее, старые элементы удаляются и новые элементы вставляются. После присваивания размер *вектора* равен числу присвоенных элементов. Например:

```
void f()
{
  vector<char> v(10, 'x');    // v.size()==10, каждый элемент имеет значение 'x'
  v.assign(5, 'a');           // v.size()==5, каждый элемент имеет значение 'a'
  // ...
}
```

Естественно, то что делает **assign()**, можно выполнить и менее прямолинейно, создав сначала подходящий вектор, а затем использовав его в операции присваивания. Вот пример:

```
void f2(vector<Book>& vh, list<Book>& lb)
{
  vector<Book> vt(lb.begin(), lb.end());
  vh = vt;
  // ...
}
```

Это, однако, и неэффективно, и некрасиво.

Может показаться, что использование конструкторов с двумя аргументами одного типа неоднозначно:

```
vector<int> v(10, 50);    // vector(size,value) или vector(iterator1,iterator2)?
                          // vector(size,value)!
```

Однако **int** — это не итератор, и реализация должна гарантировать, что на самом деле при этом вызывается

```
vector(vector<int>::size_type, const int&, const vector<int>::allocator_type&);
```

а не

```
vector(vector<int>::iterator, vector<int>::iterator, const
vector<int>::allocator_type&);
```

В стандартной библиотеке это достигается за счет подходящей перегрузки конструкторов. Таким же образом разрешаются неоднозначности при вызове **assign()** и **insert()** (§16.3.6).

16.3.5. Стековые операции

Чаще всего, мы представляем себе *вектор* в виде компактной структуры данных, доступ к элементам которой осуществляется по индексу. Однако можно игнорировать эту идею и рассматривать *вектор* как реализацию абстрактного понятия последовательности. Отсюда и из практики применения *векторов* и массивов вытекает, что для *векторов* определенный смысл имеют и стековые операции:

```
template<class T, class A = allocator<T> > class vector
{
public:
```

```
    // ...
    // стековые операции:

    void push_back(const T& x);    // добавление в конец
    void pop_back();               // удаление последнего элемента
    // ...
};
```

Эти функции работают с *вектором* как со стеком, добавляя и удаляя элементы с конца. Например:

```
void f(vector<char>& s)
{
    s.push_back('a');
    s.push_back('b');
    s.push_back('c');
    s.pop_back();
    if(s[s.size()-1] != 'b') error("impossible!");
    s.pop_back();
    if(s.back() != 'a') error("should never happen!");
}
```

Каждый раз при вызове **push_back**() вектор *s* увеличивается на один элемент, который добавляется в его конец. Таким образом, **s[s.size()-1]**, также получаемый с помощью **s.back**() (§16.3.3) — это последний добавленный (pushed) в *вектор* элемент.

В этом нет ничего необычного, если не считать применения слова «*вектор*» вместо слова «*стек*». Суффикс **_back** означает, что элементы добавляются в конец *вектора*, а не в начало. Добавление элементов в конец *вектора* может оказаться дорогостоящей операцией, поскольку это может потребовать выделения дополнительной памяти под новый элемент. Конкретные реализации обязаны обеспечить более гладкое поведение, когда операции выделения памяти происходят намного реже повторяющихся стековых операций.

Операция **pop_back**() не имеет возврата. Она лишь выполняет удаление элемента, так что если мы хотим знать, что было на вершине стека до выполнения этой операции мы сами должны озаботиться этим и предварительно посмотреть туда. Я бы не назвал стек с таким поведением моим любимым стеком (§2.5.3, §2.5.4), но он считается более эффективным, и это стандарт.

Зачем вообще могут потребоваться стековые операции над *вектором*? Наиболее очевидная причина — чтобы реализовать стек (§17.3.1), но есть и другая причина — чтобы строить *вектор* постепенно, например *вектор* геометрических точек, принимаемых из входного потока. В такой постановке задачи мы заранее не знаем, сколько точек будет получено, и поэтому не можем с самого начала определить размер *вектора* с последующим чтением значений его элементов. Вместо этого можно написать следующее:

```
vector<Point> cities;

void add_points(Point sentinel)
{
    Point buf;
```

```
    while (cin >> buf)
    {
      if (buf == sentinel) return;
      // check new point
      cities.push_back(buf);
    }
  }
```

Это гарантирует правильное расширение *вектора*. Если бы нам требовалось лишь добавлять новые точки в *вектор* **cities**, то можно было бы инициализировать его подходящим конструктором прямо из потока (§16.3.4). Но чаще всего сначала требуется несколько преобразовать входные значения и лишь затем наращивать структуру данных — **push_back**() поддерживает этот подход.

В программах на языке С в этих случаях принято использовать библиотечную функцию **realloc**(). Таким образом, **vector** (и вообще любой стандартный контейнер) обеспечивает более общую и элегантную, притом не менее эффективную альтернативу функции **realloc**().

Размер вектора, возвращаемый функцией **size**(), неявно увеличивается при вызове **push_back**(), так что переполнение (**overflow**) *векторам* не грозит (пока имеется свободная память; см. §19.4.1). Однако им грозит обратное явление — «переисчерпание» (**underflow**):

```
void f()
{
  vector<int> v;
  v.pop_back();       // состояние v становится неопределенным
  v.push_back(7);     // состояние v не определено, возможно оно недействительное
}
```

Результат «переисчерпания» не определен, но очевидная реализации функции **pop_back**() приведет к записи в область памяти, не принадлежащей *вектору*. «Переисчерпания» (как и переполнения) лучше избегать.

16.3.6. Операции над векторами, характерные для списков

Операции **push_back**(), **pop_back**() и **back**() (§16.3.5) позволяют эффективно использовать *вектор* в качестве стека. Однако иногда бывает полезно добавлять элементы в середину *вектора*, а также удалять их оттуда:

```
template<class T, class A = allocator<T> > class vector
{
public:
  // ...
  // операции, характерные для списков:

  iterator insert(iterator pos, const T& x);            // добавить x перед pos
  void insert(iterator pos, size_type n, const T& x);   // n копий x перед pos

  template<class In>                                     // In - итератор ввода (§19.2.1)
  void insert(iterator pos, In first, In last);          // вставить эл-ты из последовательности

  iterator erase(iterator pos);                          // удалить элемент в позиции pos
  iterator erase(iterator first, iterator last);         // удалить последовательность
```

```
    void clear () ;                    // удалить все элементы
    // ...
};
```

Итератор, возвращаемый функцией ***insert***(), указывает на вновь добавленный элемент. Итератор, возвращаемый функцией ***erase***(), указывает на элемент, следующий за последним удаленным элементом.

Чтобы посмотреть, как работают указанные операции, давайте проделаем несколько абсолютно формальных манипуляций с *вектором*, содержащим названия фруктов. Сначала определяем контейнер и «заселяем» его фруктами:

```
vector<string> fruit;

fruit.push_back( "peach" ) ;
fruit.push_back( "apple" ) ;
fruit.push_back( "kiwifruit" ) ;
fruit.push_back( "pear" ) ;
fruit.push_back( "starfruit" ) ;
fruit.push_back( "grape" ) ;
```

Если я разлюблю фрукты, названия которых начинаются на букву ***p***, я могу удалить соответствующие названия следующим образом:

```
sort (fruit.begin () ,fruit.end () ) ;
vector<string>::iterator p1 = find_if(fruit.begin () ,fruit.end () ,initial ('p') ) ;
vector<string>::iterator p2 = find_if(p1,fruit.end () ,initial_not ('p') ) ;
fruit.erase (p1,p2) ;
```

Другими словами, сначала сортируем *вектор*, находим первое и последнее названия фруктов, начинающиеся на ***p***, и удаляем из ***fruit*** этот интервал элементов. О том, как пишутся функции-предикаты (***predicate functions***), например ***initial***(***x***) (начальная буква есть ***x***?) или ***initial_not***(***x***) (начальная буква отличается от ***x***?), рассказывается в §18.4.2.

Операция ***erase***(***p1***,***p2***) удаляет элементы, начиная с указуемого итератором ***p1***, и вплоть до (но не включая) элемента, указуемого итератором ***p2***. Это можно проиллюстрировать графически следующим образом:

```
                                      p1              p2
fruit[] :                             |               |
                                      v               v
      apple    grape    kiwifruit    peach    pear   starfruit
```

Операция ***erase***(***p1***,***p2***) удаляет ***peach*** и ***pear***, что приводит к следующему содержимому контейнера:

```
fruit[] :
      apple    grape    kiwifruit    starfruit
```

Как всегда, затрагиваемая операцией последовательность элементов фиксируется указанием первого из них, а также указанием элемента, расположенного за последним элементом, вовлекаемым в операцию.

Было бы соблазнительно написать так:

```
vector<string>::iterator p1 = find_if(fruit.begin(),fruit.end(),initial('p'));
vector<string>::reverse_iterator p2 = find_if(fruit.rbegin(),fruit.rend(),initial ('p'));
fruit.erase(p1,p2+1);          // oops!: неверный тип
```

Однако *vector<fruit>::iterator* и *vector<fruit>::reverse_iterator* не обязаны иметь одинаковый тип, так что нельзя рассчитывать, что такой вызов *erase*() скомпилируется. Для совместного применения с прямым итератором обратный итератор нужно преобразовать явным образом:

```
fruit.erase(p1,p2.base()); // извлечь iterator из reverse_iterator (§19.2.5)
```

Удаление элемента из вектора изменяет его размер; при этом элементы, располагающиеся во след удаляемым, копируются на освобождающиеся позиции. В нашем примере *fruit.size*() возвращает число *4*, а *starfruit*, ранее доступная как *fruit*[*5*], становится *fruit*[*3*].

Естественно, можно удалять и единственный элемент. В этом случае нужно с помощью итератора указать один лишь этот элемент. Вот соответствующий пример:

```
fruit.erase(find(fruit.begin(),fruit.end(),"starfruit"));
fruit.erase(fruit.begin()+1);
```

Здесь удаляются *starfruit* и *grape*, после чего в контейнере *fruit* остаются два элемента:

```
fruit[]:
    apple    kiwifruit
```

В вектор также можно вставлять элементы. Например:

```
fruit.insert(fruit.begin()+1,"cherry");
fruit.insert(fruit.end(),"cranberry");
```

Новый элемент вставляется перед указуемым элементом с одновременным сдвигом всех элементов (включая указуемый) для освобождения необходимого места. В итоге контейнер *fruit* получит следующее содержимое:

```
fruit[]:
    apple    cherry    kiwifruit    cranberry
```

Отметим, что *f.insert*(*f.end*(),*x*) эквивалентно *f.push_back*(*x*).

Можно вставить целую последовательность элементов:

```
fruit.insert(fruit.begin()+2,citrus.begin(),citrus.end());
```

Если контейнер *citrus* имеет содержимое

```
citrus[]:
    lemon    grapefruit    orange    lime
```

то *вектор fruit* станет таким:

```
fruit[]:
    apple cherry lemon grapefruit orange lime kiwifruit cranberry
```

Элементы контейнера ***citrus*** копируются в вектор ***fruit*** функцией ***insert***(), причем содержимое ***citrus*** остается при этом без изменений.

Очевидно, что операции ***insert***() и ***erase***() более универсальны, чем операции, затрагивающие лишь концевое содержимое *вектора* (§16.3.5). Но они и более дорогие операции, так как, например, при вставке элемента может потребоваться перемещать часть элементов на новые места. Если для конкретной задачи типичны вставки и удаления элементов, то лучше выбрать список вместо вектора. Список ***list*** оптимизирован для операций ***insert***() и ***erase***(), а не для операций индексирования (§16.3.3).

Вставка элементов и их удаление из вектора (но не списка или ассоциированного контейнера вроде ***map***) потенциально передвигает элементы. Следовательно, итератор, указывавший на какой-то элемент вектора, после операций ***insert***() или ***erase***() может уже показывать на другой элемент вектора, или даже вообще не на элемент вектора. Никогда не обращайтесь к элементу по недействительному итератору — результат такой операции не определен и последствия, скорее всего, будут катастрофическими. В частности, остерегайтесь использовать итератор, отмечавший место вставки; ***insert***() делает его недействительным. Например:

```
void duplicate_elements(vector<string>& f)
{
  for(vector<string>::iterator p = f.begin(); p!=f.end(); ++p)
    f.insert(p, *p);      // Не надо!
}
```

Подумайте над этим (§16.5[15]). Реализация вектора передвинет все элементы или, по крайней мере, все элементы после ***p*** для того, чтобы освободить место для нового элемента.

Операция ***clear***() удаляет из контейнера все элементы. Таким образом, ***c.clear***() — это краткая запись для ***c.erase***(***c.begin***(), ***c.end***()). После ***c.clear***() вызов ***c.size***() вернет *нуль*.

16.3.7. Адресация элементов

Чаще всего операции ***erase***() или ***insert***() применяются к хорошо известным местам в контейнере (например, ***begin***() или ***end***()), к результатам операций поиска (например, ***find***()) или к позициям, определяемым итерационно. В таких случаях итератор указывает на соответствующий элемент. К элементам *векторов* мы также часто обращаемся по индексу. В связи с этим возникает вопрос, как нам получить итератор, необходимый операциям ***insert***() или ***erase***(), для элемента вектора ***c*** с индексом *7*? Поскольку это седьмой элемент вектора после его начала, ответом будет выражение ***c.begin***()+7. Другие альтернативы, кажущиеся приемлемыми по аналогии с массивами, вполне могут оказаться ошибочными (а значит их следует избегать). Рассмотрим пример:

```
template<class C> void f(C& c)
{
  c.erase(c.begin()+7);    // ok (если итераторы для c поддерживают + (см. §19.2.1))
  c.erase(&c[7]);          // не универсальное решение
  c.erase(c+7);            // error: прибавление 7 к контейнеру бессмысленно
  c.erase(c.back());       // error: c.back() это ссылка, а не итератор
```

```
    c.erase(c.end()-2);                 // ok (предпоследний элемент)
    c.erase(c.rbegin()+1);              // error: reverse iterator и iterator - разные типы
    c.erase((c.rbegin()+2).base());     // туманно, но ok (см. §19.2.5)
}
```

Самая соблазнительная альтернатива, `&c[7]`, работает для очевидных реализаций *вектора*, в которых `c[7]` ссылается на элемент и его адрес есть действительный итератор. Однако с может оказаться контейнером, для которого итераторы не являются простыми указателями на элементы. Например, операция индексирования для ассоциативных массивов map (§17.4.1.3) имеет возврат типа ***mapped_type***&, а не ссылку на элемент (***value_type***&).

Не все контейнеры поддерживают операцию + для своих итераторов. Например, ***list*** не допускает даже ***c.begin***()+7. Если вам на самом деле необходимо увеличить ***list::iterator*** на 7, примените многократно операцию ++, или вызовите ***advance***() (§19.2.1).

Мнимые альтернативы *c+7* и ***c.back***() — просто ошибки в типах. Контейнер не является числовой переменной, к которой можно прибавить 7, а ***c.back***() — это элемент со значением вроде ***pear*** (груша), не идентифицирующий положение «груши» в контейнере ***c***.

16.3.8. Размер и емкость

До сих пор мы описывали вектор с минимальным упоминанием вопросов, связанных с управлением памятью — вектор растет по мере необходимости, и все. Обычно этого достаточно. Но можно задать и явный вопрос о том, как именно вектор получает память, и даже прямо вмешаться в этот процесс. Вот некоторые операции вектора:

```
template <class T, class A = allocator<T> > class vector
{
public:
    // ...
    // capacity - емкость:

    size_type size() const;                   // число элементов
    bool empty() const {return size()==0;}
    size_type max_size() const;               // размер максимально длинного вектора

    void resize(size_type sz, T val = T());   // добавляемые эл-ты иниц-ся значением val

    size_type capacity() const;   // размер выделенной (в элементах) памяти
    void reserve(size type n);    // выделяет память для n элементов без инициализации;
                                  // генерирует исключение length_error если n>max_size()

    // ...
};
```

В любой момент времени вектор «знает» число своих элементов. Это число возвращается функцией ***size***() и может изменяться вызовом ***resize***(). Таким образом, пользователь всегда может определить размер вектора и изменить его, если это нужно. Например:

```
class Histogram
{
```

```
    vector<int> count;

public:
    Histogram(int h) : count(max(h,8)) { }
    void record (int i);
    // ...
};

void Histogram::record(int i)
{
    if(i<0) i=0;
    if(count.size() <= i) count.resize(i+i);   // выделяем достаточно памяти
    count[i]++;
}
```

Применение ***resize*()** к вектору очень похоже на применение функции ***realloc*()** из библиотеки языка С ко встроенным массивам, динамически выделяемым в свободной памяти (куче).

Когда размеры вектора изменяются под большее (или меньшее) число элементов, элементы при этом могут быть перемещены в новую область памяти. Поэтому идея хранить указатели на элементы вектора, который может подвергнуться операции ***resize*()**, весьма плоха, ибо после этой операции старые адреса элементов могут стать недействительными. Лучше хранить индексы элементов. Обратите также внимание на то, что операции ***push_back*()**, ***pop_back*()**, ***insert*()** и ***erase*()** изменяют размер вектора неявно.

Помимо объема памяти, необходимого для хранения элементов, конкретная реализация может выделять дополнительную память для потенциального расширения. Программист, полагающий, что такое расширение вероятно, может вызовом функции ***reserve*()** зарезервировать заданное количество памяти под будущее расширение. Например:

```
struct Link
{
    Link* next;
    Link(Link* n =0) : next(n) { }
    // ...
};

vector<Link> v;

void chain (size_t n)   // заполняет v n элементами Link так, чтобы каждый Link
                        // указывал на предшествующий
{
    v.reserve(n);
    v.push_back(Link(0));
    for(int i=1; i<n; i++) v.push_back(Link(&v[i-1]));
    // ...
}
```

Вызов ***v.reserve*(*n*)** гарантирует, что при увеличении размера вектора ***v*** не потребуется никакого дополнительного выделения памяти до тех пор, пока ***v.size*()** не превысит ***n***.

Предварительное резервирование памяти имеет два преимущества. Во-первых, даже самая бесхитростная реализация может за один раз выделить достаточный объем памяти вместо того, чтобы многократно требовать дополнительного выделения памяти. А во-вторых, в ряде случаев существует еще и логическое преимущество, перевешивающее соображения эффективности. По мере роста контейнера элементы могут перемещаться на новые места в памяти. Поэтому в представленном выше примере осмысленность (неизменность) связей между элементами, установленными в процессе роста вектора, гарантируется лишь предварительным вызовом ***reserve***(), предохраняющим от выделения памяти по мере его роста. Итак, в некоторых случаях помимо повышения эффективности предварительное резервирование памяти вообще обеспечивает корректность решения задачи.

Дополнительно, предварительное выделение памяти гарантирует, что потенциальное исчерпание памяти и дорогостоящее перемещение элементов произойдет лишь в предсказуемый момент. Для программ реального времени это крайне важно.

Заметим, что функция ***reserve***() не изменяет размера вектора, так что ей не надо инициализировать новые элементы. В этом отношении она полностью отличается от функции ***resize***().

Так же, как функция ***size***() возвращает текущее число элементов, функция ***capacity***() возвращает текущее значение зарезервированных мест: ***c.capacity***() — ***c.size***() дает число элементов, которые еще можно добавить без необходимости выделения дополнительной памяти.

Уменьшение размера вектора не уменьшает его емкости. Это просто увеличивает число свободных мест, пригодных для будущего увеличения вектора. Чтобы вернуть память системе, нужно выполнить небольшой трюк:

```
vector<int> tmp = v;    // копирование v с умолчательной емкостью
v.swap(tmp);            // теперь v получает умолчательную емкость (см. §16.3.9)
                        // а память под ее предыдущую емкость освободится,
                        // когда tmp выйдет из области видимости
```

Вектор получает необходимую для его элементов память с помощью вызова функций-членов его аллокатора (который предоставляется в виде шаблонного параметра). Аллокатор по умолчанию, называемый ***allocator*** (§19.4.1), использует для выделения памяти операцию ***new***, так что если свободной памяти больше не останется, будет сгенерировано исключение ***bad_alloc***. Другие распределители памяти (аллокаторы) могут реализовывать иные стратегии поведения (см. §19.4.2).

Функции ***reserve***() и ***capacity***() уникальны для контейнера ***vector*** и подобных ему контейнеров. Списки ничего подобного не предоставляют.

16.3.9. Другие функции-члены

Многие алгоритмы, включая важные алгоритмы сортировки, нуждаются в операции обмена элементов местами. Очевидным способом такого обмена (§13.5.2) является копирование элементов. Однако в типичном случае ***vector*** реализуется с помощью дескриптора доступа к его элементам (§13.5, §17.1.3). Поэтому наиболее эффективный обмен двух *векторов* выполняется путем обмена дескрипторов; это делает функция ***vector::swap***(). Для многих важных случаев достигаемый при этом выигрыш во времени может составлять несколько порядков:

```
template<class T, class A = allocator<T> > class vector
{
public:
  // ...
  void swap (vector&) ;
  allocator_type get_allocator () const;
};
```

Функция ***get_allocator***() позволяет программисту получить доступ к аллокатору вектора (§16.3.1, §16.3.4). Это позволяет, например, выделять память под данные программы, использующей вектор, тем же способом, что и память под сам *вектор* (§19.4.1).

16.3.10. Вспомогательные функции (helper functions)

Два *вектора* можно сравнивать операциями == и <:

```
template<class T, class A>
bool std::operator== (const vector<T,A>& x, const vector<T,A>& y) ;

template<class T, class A>
bool std::operator< (const vector<T,A>& x, const vector<T, A>& y) ;
```

Два вектора ***v1*** и ***v2*** считаются равными, если ***v1.size***()==***v2.size***() и ***v1***[***n***]==***v2***[***n***] для каждого действительного индекса ***n***. Аналогично, операция < выполняется в соответствии с лексикографическим упорядочением. Иными словами, эту операцию для *векторов* можно было бы определить следующим образом:

```
template<class T, class A>
inline bool std::operator< (const vector<T, A>& x, const vector<T,A>& y)
{
  return lexicographical_compare (x.begin () , x.end() , y.begin () , y.end() ) ; //см. §18.9
}
```

Это означает, что ***x*** меньше ***y***, если первый ***x***[***i***], не равный соответствующему ***y***[***i***], меньше, чем ***y***[***i***], или ***x.size***()<***y.size***() при равенстве всех ***x***[***i***] соответствующим ***y***[***i***].

Стандартная библиотека предоставляет также операции !=, <=, > и >= с определениями, аналогичными таковым для == и <.

Поскольку ***swap***() является функцией-членом, она вызывается как ***v1.swap***(***v2***). Поскольку не любой тип контейнера имеет функцию-член ***swap***(), в обобщенных алгоритмах применяется более традиционный вызов ***swap***(***a***, ***b***). Чтобы это работало и для *векторов*, стандартная библиотека предоставляет следующую специализацию:

```
template<class T, class A> void std::swap (vector<T,A>& x, vector<T,A>& y)
{
  x.swap (y) ;
}
```

16.3.11. Специализация vector<bool>

Специализация ***vector<bool>*** (§13.5) предоставляется как очень компактный вектор элементов типа ***bool***. Поскольку любая переменная типа ***bool*** адресуема, под нее

отводится один байт памяти. Однако можно эффективно реализовать ***vector<bool>***, отведя под каждый элемент ровно один бит.

Обычные операции над векторами работают с сохранением своего смысла и над векторами типа ***vector<bool>***. В частности, индексирование и итерирование выполняются вполне традиционно. Например:

```
void f(vector<bool>& v)
{
  for (int i=0; i<v.size(); ++i) cin>>v[i];              // индексирование

  typedef vector<bool>::const_iterator VI;
  for (VI p=v.begin(); p != v.end(); ++p) cout<<*p;     // применение итераторов
}
```

Чтобы этого достичь, реализация должна имитировать доступ к битам при помощи традиционного синтаксиса. Поскольку указателем нельзя адресовать единицу памяти, меньшую байта, то ***vector<bool>::iterator*** не может быть указателем. В частности, нельзя трактовать итератор контейнеров типа ***vector<bool>*** как ***bool****:

```
void f{vector<bool>& v)
{
  bool* p = v.begin();   // error: несоответствие типов
  // ...
}
```

Техника адресации битов в общих чертах рассматривается в §17.5.3.

Стандартная библиотека также предоставляет специальный контейнер ***bitset*** (битовое поле) как множество булевых (логических) значений вместе с набором булевых операций (§17.5.3).

16.4. Советы

1. Используйте средства стандартной библиотеки для поддержки переносимости; §16.1.
2. Не пытайтесь переопределять средства стандартной библиотеки; §16.1.2.
3. Не верьте тому, что стандартная библиотека есть самое лучшее решение для любых задач.
4. Создавая новое средство, подумайте, нельзя ли его реализовать средствами стандартной библиотеки; §16.3.
5. Помните, что средства стандартной библиотеки определены в пространстве имен ***std***; §16.1.2.
6. Объявляйте средства стандартной библиотеки, включая стандартный заголовочный файл (а не явным определением); §16.1.2.
7. Пользуйтесь преимуществами поздней абстракции; §16.2.1.
8. Избегайте «жирных» интерфейсов; §16.2.2.
9. Явным циклам «из конца в начало» предпочитайте алгоритмы с обратными итераторами; §16.3.2.

10. Используйте ***base***() для извлечения ***iterator*** из ***reverse_iterator***; §16.3.2.
11. Передавайте контейнеры по ссылке; §16.3.4.
12. Для ссылки на элементы контейнеров используйте итераторные типы, такие как ***list<char>::iterator***, а не указатели; §16.3.1.
13. Используйте *константные* итераторы, когда не нужно изменять значения элементов контейнера; §16.3.1.
14. Прямо или косвенно используйте ***at***() в случае необходимости в проверке диапазона индексов; §16.3.3.
15. Используйте ***push_back***() или ***resize***() для контейнеров вместо ***realloc***() для массивов; §16.3.5.
16. Не используйте итераторов, адресующих элементы вектора, подвергнутого операции ***resize***(); §16.3.8.
17. Используйте ***reserve***() для того, чтобы обеспечивать итераторам действительность их значений на длительное время; §16.3.8.
18. При необходимости используйте ***reserve***() для гарантии эффективности и корректности работы; §16.3.8.

16.5. Упражнения

Решение некоторых задач можно найти, посмотрев реализацию стандартной библиотеки. Сделайте себе полезное одолжение: не пытайтесь смотреть сторонний код до того, как вы сами попробовали решить задачу.

1. (*1.5) Создайте ***vector<char>***, содержащий буквы в алфавитном порядке. Распечатайте элементы этого вектора в прямом и обратном порядке.
2. (*1.5) Создайте ***vector<string>*** и считайте в него названия фруктов из входного потока ***cin***. Отсортируйте содержимое вектора и распечатайте его.
3. (*1.5) Используя *вектор* из §16.5[2], запрограммируйте цикл распечатки всех фруктов, имена которых начинаются на букву ***a***.
4. (*1) Используя *вектор* из §16.5[2], запрограммируйте цикл для удаления из вектора названий всех фруктов, имена которых начинаются на букву ***a***.
5. (*1) Используя *вектор* из §16.5[2], запрограммируйте цикл для удаления из *вектора* всех цитрусовых.
6. (*1.5) Используя *вектор* из §16.5[2], запрограммируйте цикл для удаления из *вектора* всех фруктов, которые вы не любите.
7. (*2) Завершите классы ***Vector***, ***List*** и ***Itor*** из §16.2.1.
8. (*2.5) Отталкиваясь от класса ***Itor***, подумайте, как обеспечить прямые и обратные итераторы, итераторы для контейнера, содержимое которого может меняться в процессе итерации и итераторы для работы поверх контейнеров с неизменным содержимым. Организуйте этот набор итераторов так, чтобы пользователь мог взаимозаменяемо использовать итераторы, обеспечивающие достаточную для алгоритма функциональность. Минимизируйте повторение кода при реализации этих контейнеров. Какие еще типы итераторов

могут потребоваться пользователю? Перечислите достоинства и недостатки вашего подхода.

9. (*2) Завершите классы ***Container***, ***Vector*** и ***List*** из §16.2.2.
10. (*2.5) Сгенерируйте ***10000*** однородно распределенных на интервале от ***0*** до ***1023*** чисел и сохраните их в: а) стандартном библиотечном контейнере ***vector***; b) контейнере ***Vector*** из §16.5[7]; с) контейнере ***Vector*** из §16.5[9]. Во всех случаях вычислите среднее арифметическое элементов контейнеров. Засеките время. Оцените, измерьте и сравните между собой объемы потребления памяти для трех этих стилей вектора.
11. (*1.5) Напишите такой итератор, чтобы ***Vector*** из §16.2.2 мог использоваться в стиле контейнера из §16.2.1.
12. (*1.5) Напишите класс, производный от ***Container*** так, чтобы ***Vector*** из §16.2.1 мог использоваться в стиле контейнера из §16.2.2.
13. (*2) Напишите такие классы, чтобы ***Vector*** из §16.2.1 и ***Vector*** из §16.2.2 могли использоваться как стандартные контейнеры.
14. (*2) Напишите шаблон, реализующий контейнер с теми же функциями-членами и теми же типами, что и стандартный ***vector***, для существующего (нестандартного) контейнерного типа. Не модифицируйте при этом существующий контейнерный тип. Как вы поступите с его функциональностью, не совпадающей со стандартной?
15. (*1.5) Кратко опишите возможное поведение функции ***duplicate_elements***() из §16.3.6 для ***vector<string>***, состоящего из трех элементов: ***don't do this*** (не делай этого).

17

Стандартные контейнеры

Теперь самое время поставить вашу работу
на прочный теоретический фундамент.
— Сэм Морган

Стандартные контейнеры — обзор операций и контейнеров — эффективность — внутреннее представление — требования к элементам контейнеров— последовательные контейнеры — ***vector*** — ***list*** — ***deque*** — адаптеры — стек — очереди — очереди с приоритетами — ассоциативные контейнеры — ассоциативные массивы — сравнения — контейнеры ***multimap*** — контейнеры ***set*** — контейнеры ***multiset*** — «почти контейнеры» — битовые поля ***bitset*** — встроенные массивы — хэш-таблицы — реализация ***hash_map*** — советы — упражнения.

17.1. Стандартные контейнеры

В стандартной библиотеке определяются два вида контейнеров — *последовательные контейнеры* (*sequences*) и *ассоциативные контейнеры* (*associative containers*). Все последовательные контейнеры похожи на ***vector*** (§16.3). Если не сказано обратное, типы и функции, упоминавшиеся в связи с *векторами*, могут с тем же успехом использоваться и для других последовательных контейнеров. В то же время ассоциативные контейнеры дополнительно обеспечивают доступ к своим элементам на основе ключей (§3.7.4).

Встроенные массивы (§5.2), объекты типа ***string*** (глава 20) и ***valarray*** (§22.4), а также битовые поля ***bitset*** (§17.5.3) содержат элементы, и тоже могут трактоваться как контейнеры. Однако они не являются в точном архитектурном смысле стандартными контейнерами. Если бы это было так, то они не удовлетворяли бы своему первичному (главному) предназначению. Например, встроенные массивы содержали бы в таком случае свой размер, а это препятствовало бы их совместимости со встроенными массивами языка С.

Ключевая идея стандартных контейнеров состоит в их взаимозаменяемости в тех случаях, где это имеет смысл. Пользователь может выбирать между ними на

основе вопросов эффективности и наличия специализированных операций. Например, если в задаче часто требуется поиск по ключу, то стоит применить ассоциативный контейнер ***map*** (§17.4.1). С другой стороны, если преобладают типичные для списков операции, то стоит применить ***list*** (§17.2.2). Если большая часть вставок и удалений элементов осуществляется на концах контейнера, то лучше всего подойдут ***deque*** (двусторонняя очередь, §17.2.3), ***stack*** (стек, §17.3.1) или ***queue*** (очередь, §17.3.2). Кроме того, пользователь может построить свой собственный контейнер в согласии с архитектурой и приемами построения стандартных контейнеров (§17.6). По умолчанию, в этом случае лучше использовать ***vector*** (§16.3), ибо он реализован столь универсально, что его можно применять в самых разных контекстах.

Идея унифицированной работы с разными типами контейнеров (в более общем контексте — с различными источниками информации) приводит к понятию обобщенного программирования (§2.7.2, §3.8). Для поддержки этой идеи стандартная библиотека содержит множество обобщенных алгоритмов (глава 18), которые позволяют программисту абстрагироваться от конкретных деталей индивидуальных контейнеров.

17.1.1. Обзор контейнерных операций

В этом разделе перечислены почти все общие для стандартных контейнеров типы, итераторы, функции доступа и другие операции. За большими подробностями обращайтесь к стандартным заголовочным файлам (<***vector***>, <***list***>, <***map***> и др.; см. §16.1.2).

Типы (§16.3.1)	
value_type	Тип элемента
allocator_type	Тип аллокатора
size_type	Тип индексов, счетчиков и т.д.
difference_type	Тип разности итераторов
iterator	Ведет себя подобно ***value_type******
const_iterator	Ведет себя подобно const ***value_type******
reverse_iterator	Рассматривает контейнер в обратном порядке; как ***value_type******
const_reverse_iterator	Рассматривает контейнер в обратном порядке; как ***const value_type******
reference	***value_type***&
const_reference	***const value_type***&
key_type	Тип ключа (для ассоциативных контейнеров)
mapped_type	Тип ***mapped_value*** (для ассоциативных контейнеров)
key_compare	Тип критерия сравнения (для ассоциативных контейнеров)

Контейнер можно последовательно просмотреть в определенном порядке с помощью его типа ***iterator***, или в обратном порядке (с помощью обратных итерато-

ров). Для ассоциативных контейнеров порядок просмотра основывается на его критерии сравнения (по умолчанию это <):

Итераторы (§16.3.2)	
begin ()	Указывает на первый элемент
end ()	Указывает на элемент, следующий за последним
rbegin ()	Указывает на первый элемент обратной последовательности
rend ()	Указывает на элемент, следующий за последним в обратной последовательности

К некоторым элементам возможен прямой доступ:

Доступ к элементам (§16.3.3)	
front ()	Первый элемент
back ()	Последний элемент
[]	Индексирование, доступ без проверки (не для списка)
at ()	Индексирование, доступ с проверкой (для ***vector*** и ***deque***)

Векторы и двусторонние очереди обеспечивают эффективные операции в конце последовательности своих элементов. Кроме того, списки и двусторонние очереди обеспечивают эффективные операции в начале последовательности своих элементов.

Операции со стеком и очередями (§16.3.5, §17.2.2.2)	
push_back ()	Добавить в конец
pop_back ()	Удалить последний элемент
push_front ()	Добавить новый первый элемент (для ***list*** и ***deque***)
pop_front ()	Удалить первый элемент (для ***list*** и ***deque***)

Контейнеры предоставляют операции, типичные для списков:

Операции со списками (§16.3.6)	
insert (***p***, ***x***)	Добавить x перед p.
insert (***p***, ***n***, ***x***)	Добавить ***n*** копий ***x*** перед ***p***
insert (***p***, ***first***, ***last***)	Добавить элементы из [***first*** : ***last***[перед ***p***
erase (***p***)	Удалить элемент в позиции ***p***
erase (***first***, ***last***)	Удалить элементы [***first*** : ***last***[
clear ()	Удалить все элементы

В приложении Е обсуждается поведение контейнеров в случае, когда операции аллокаторов или операции над элементами генерируют исключения.

Контейнеры предоставляют операции получения числа элементов и некоторые другие операции:

Другие операции (§16.3.8, §16.3.9, §16.3.10,)	
size ()	Число элементов
empty ()	Контейнер пуст?
max_size ()	Размер максимально возможного контейнера
capacity ()	Память, выделенная под *vector*
reserve ()	Резервирует память для потенциальных расширений (только для контейнера *vector*)
resize ()	Изменяет размер контейнера (для *vector*, *list* и *deque*)
swap ()	Обмен элементами у двух контейнеров
get_allocator ()	Предоставляет копию контейнерного аллокатора
==	Содержимое контейнеров совпадает?
!=	Содержимое контейнеров различно?
<	Один контейнер лексикографически предшествует второму?

Контейнеры реализуют множественные варианты конструкторов и операций присваивания:

Конструкторы и т.д. (§16.3.4)	
container ()	Пустой контейнер
container (*n*)	*n* элементов с умолчательными значениями (не для ассоциативных контейнеров)
container (*n*,*x*)	*n* копий *x* (не для ассоциативных контейнеров)
container (*first*, *last*)	Инициализируется элементами из [*first*:*last*[
container (*x*)	Копирующий конструктор (инициализируется элементами из *x*)
~*container* ()	Уничтожить контейнер и все его элементы

Присваивания (§16.3.4)	
operator= (*x*)	Копирующее присваивание; элементы берутся из контейнера *x*
assign (*n*, *x*)	Присвоить *n* копий *x* (не для ассоциативных контейнеров)
assign (*first*, *last*)	Присваивание из [*first*:*last*[

Ассоциативные контейнеры обеспечивают поиск на основе ключей:

Операции ассоциативных контейнеров (§17.4.1)	
operator [ ] (***k***)	Доступ к элементу с ключом ***k*** (для контейнеров с уникальными ключами)
find (***k***)	Найти элемент с ключом ***k***
lower_boud (***k***)	Найти первый элемент с ключом ***k***
upper_bound (***k***)	Найти первый элемент с ключом, большим ***k***
equal_range (***k***)	Найти ***lower_bound*** и ***upper_bound*** для элемента с ключом ***k***
key_comp ( )	Копия объекта сравнения ключей
value_comp ( )	Копия объекта сравнения ***mapped_value***

Кроме этих общих операций большинство контейнеров предоставляют также и специализированные операции.

17.1.2. Краткий обзор контейнеров

Приведем краткую сводную информацию по стандартным контейнерам в следующей табличной форме:

Операции стандартных контейнеров					
	[]	List операции	Front операции	Back (Stack) операции	Итераторы
	§ 16.3.3 § 17.4.1.3	§ 16.3.6 § 20.3.9	§ 17.2.2.2 § 20.3.9	§ 16.3.5 § 20.3.12	§ 19.2.1
vector	const	O(n)+		const+	Ran
list		const	const	const	Bi
deque	const	O(n)	const	const	Ran
stack				const	
queue			const	const	
priority_queue			O(log(n))	O(log(n))	
map	O(log(n))	O(log(n))+			Bi
multimap		O(log(n))+			Bi
set		O(log(n))+			Bi
multiset		O(log(n))+			Bi
string	const	O(n)+	O(n)+	const+	Ran
array	const				Ran
valarray	const				Ran
bitset	const				

В колонке ***Iterators*** аббревиатура ***Ran*** означает итератор произвольного доступа, а ***Bi*** — двунаправленный итератор; операции с двунаправленными итераторами являются подмножеством операций с итераторами произвольного доступа (§19.2.1). Другие записи в этой таблицы означают степень эффективности операций. Запись ***const*** означает, что операция занимает время, не зависящее от числа элементов контейнера. По-другому, это же можно записать в виде ***O(1)***. Запись ***O(n)*** означает, что операция занимает время, пропорциональное числу участвующих в операции элементов. Суффикс + означает, что в ряде случаев операция может потребовать дополнительных затрат времени. Например, вставка элемента в список имеет фиксированную цену (поэтому она помечена как ***const***), в то время как для векторов она приводит к перемещению всех элементов, начиная с места вставки (что соответствует ***O(n)***). Но иногда при этом приходится перемещать все элементы (поэтому я добавил +). Обозначение с большой буквой ***O*** абсолютно традиционное. Знак + я добавил для программистов, которым помимо средней оценки быстродействия важна еще и предсказуемость. Для ***O(n)***+ принят термин *amortized linear time* (*линейное время с амортизацией*).

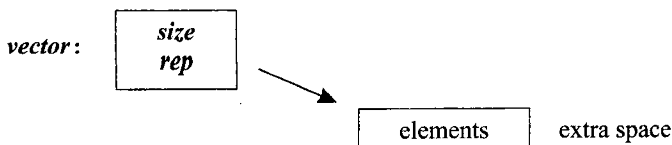
Очевидно, что если ***const*** очень велико, то это может и превысить оценку ***O(n)***. В то же время, если число элементов велико, то ***const*** будет значить «дешево», ***O(n)*** будет значить дорого, а ***O(log(n))*** — «более-менее дешево». Для умеренно больших ***n*** оценка ***O(log(n))*** ближе к константе, чем к оценке ***O(n)***. Для тех, кому цена операции очень важна, лучше присмотреться к этому повнимательнее, например поточнее понять, какие элементы формируют число ***n***. Наконец отметим, что никакие перечисленные в таблице базовые операции не являются «слишком дорогими», то есть среди них нет операций порядка ***O(n*n)*** или хуже того.

За исключением ***string***, перечисленные стоимостные оценки соответствуют требованиям стандарта. Оценки для ***string*** — это мои предположения. Оценки для ***stack*** и ***queue*** соответствуют умолчательной реализации с применением ***deque*** (§17.3.1, §17.3.2).

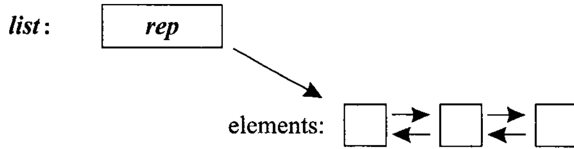
Приведенные оценки сложности и цены являются оценками сверху. Они позволяют пользователю прикинуть, что можно ожидать от разных реализаций. Для особо важных случаев некоторые реализации могут предоставить варианты получше.

17.1.3. Внутреннее представление

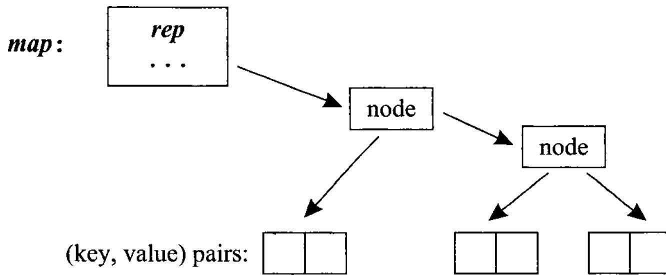
Стандарт не определяет для контейнеров никаких внутренних представлений (***representation***, сокращенно — ***rep***). Вместо этого, он специфицирует открытый интерфейс и некоторые требования к сложности. Чтобы удовлетворить стандарту, разработчики применяют подходящие и тщательно оптимизированные варианты. Контейнер в общем случае представляется некоторой структурой данных, содержащей элементы, доступной через дескриптор, содержащий размер и емкость контейнера. Для контейнера ***vector***, структура данных под элементы является непрерывной областью памяти, фактически массивом:



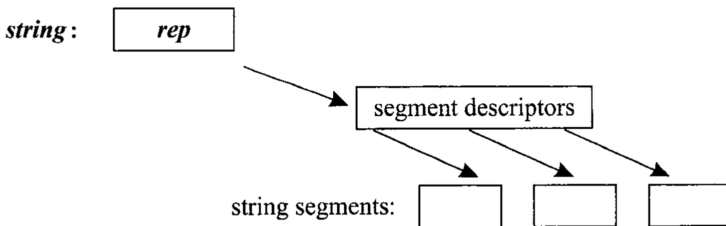
Для списков структура данных в типичном представлении является набором связей, указывающих на элементы:



Ассоциативный контейнер map наиболее вероятно представим сбалансированным деревом узлов, указывающих на пары (ключ,значение) (key,value):



Строки могут реализовываться так, как описано в §11.12, или как последовательность массивов, каждый из которых содержит небольшой набор символов:



17.1.4. Требования к элементам контейнеров

Элементы в контейнерах — это копии вставляемых объектов. Таким образом, чтобы объект мог стать элементом контейнера, он должен иметь тип, позволяющий реализации контейнера скопировать объект. Контейнер может копировать объекты посредством копирующего конструктора или операции присваивания; в обоих случаях копия должна быть эквивалентна исходному объекту. Грубо говоря, это означает, что любая ваша проверка на равенство значений объектов должна констатировать равенство копии и оригинала. Другими словами, копирование элементов должно работать весьма похоже на копирование встроенных типов (включая указатели). Например, следующий код

```
X& X::operator=(const X& a)          // подходящая операция присваивания
{
  // копирование всех членов a в *this
  return *this;
}
```

делает *X* подходящим типом элементов стандартных контейнеров, но

```
void Y::operator=(const Y& a)        // не подходящая операция присваивания
{
  // обнуление всех членов a
}
```

того же самого в отношении типа *Y* не достигает: для него операция присваивания не имеет традиционной семантики и не имеет принятого возвращаемого типа.

Некоторые нарушения правил для стандартных контейнеров обнаруживаются компилятором, но те нарушения, которые компилятором не выявляются, могут приводить к непредсказуемому поведению. Например, операция присваивания, генерирующая исключения, может оставить некоторые элементы недокопированными (в промежуточном состоянии), или вообще привести их в некорректное состояние, что вызовет серьезные проблемы впоследствии. Такая операция присваивания, вообще говоря, сама плохо спроектирована (§14.4.6.1, приложение Е).

Когда копирование элементов для их помещения в контейнер — это не то, что требуется, альтернативой является помещение в контейнер указателей на объекты. Наиболее типичными примерами являются полиморфные типы (§2.5.4, §12.2.6). Например, для сохранения полиморфного поведения мы применяем **vector<Shape*>**, а не **vector<Shape>**.

17.1.4.1. Операция сравнения "<"

Ассоциативные контейнеры требуют, чтобы их элементы могли быть упорядочены. То же относится и ко многим операциям, которые применяются к контейнерам (например, **sort()**). По умолчанию, для упорядочения элементов используется операция <. Если же эта операция не подходит, то программист сам предоставляет альтернативу (§17.4.1.5, §18.4.2). Критерий упорядочения должен обеспечить так называемое *строгое слабое упорядочение* (*strict weak ordering*). Неформально это означает, что «меньше» и «равно» должны быть транзитивными. То есть для критерия упорядочения **cmp**:

1. **cmp(x,x)** есть «ложь».
2. Если **cmp(x,y)** — «истина» и **cmp(y,z)** — «истина», то **cmp(x,z)** — тоже «истина».
3. Определим **equiv(x,y)** как **!(cmp(x,y)||cmp(y,x))**. Тогда, если **equiv(x,y)** — «истина» и **equiv(y,z)** — «истина», то и **equiv(x,z)** тоже «истина».

Рассмотрим:

```
template<class Ran> void sort(Ran first, Ran last);               // используем < для сравнения
template<class Ran, class Cmp> void sort(Ran first, Ran last, Cmp cmp); // используем cmp
```

Первая из представленных версий использует операцию <, а вторая — пользовательскую функцию сравнения **cmp()**. Допустим, мы хотим отсортировать контей-

нер ***fruit***, используя нечувствительное к регистру букв сравнение. Мы можем это сделать, определив класс функциональных объектов (§11.9, §18.4), выполняющий сравнение, будучи вызванным для пары значений типа ***string***:

```
class Nocase    // case-insensitive string compare
{
public:
  bool operator() (const string&, const string&) const;
};

bool Nocase::operator() (const string& x, const string& y) const
// return true if x is lexicographically less than y, not taking case into account
{
  string::const_iterator p = x.begin();
  string::const_iterator q = y.begin();

  while(p!=x.end() && q!=y.end() && toupper(*p)==toupper(*q))
  {
    ++p;
    ++q;
  }

  if(p == x.end()) return q != y.end();
  if(q == y.end()) return false;
  return toupper(*p) < toupper(*q);
}
```

Теперь мы можем вызвать ***sort()***, используя данный критерий сравнения. Например, пусть контейнер ***fruit*** изначально имеет следующее содержимое:

```
fruit:
    apple    pear    Apple    Pear    lemon
```

Тогда вызвав ***sort(fruit.begin(), fruit.end(), Nocase())***, получим

```
fruit:
    Apple    apple    lemon    Pear    pear
```

в то время как обычная сортировка ***sort(fruit.begin(), fruit.end())*** дала бы иной результат:

```
fruit:
    Apple    Pear    apple    lemon    pear
```

в предположении, что в заданной кодировке заглавные буквы идут раньше строчных.

Имейте в виду, что операция < для С-строк (то есть для типа ***char****) не дает лексикографического упорядочения (§13.5.2). Из-за этого ассоциативные контейнеры работают неожиданно для многих пользователей, когда в качестве ключей используются С-строки. Чтобы заставить их работать надлежащим образом, для сравнения С-строк нужно организовать лексикографическое сравнение, например, следующим образом:

```
struct Cstring_less
{
  bool operator() (const char* p, const char* q) const
```

```
    {return strcmp(p,q)<0;}
};

map<char*, int, Cstring_less> m;    // здесь map использует strcmp() для сравнения
                                    // ключей типа const char*
```

17.1.4.2. Другие операции сравнения

Итак, по умолчанию контейнеры и алгоритмы используют операцию <, чтобы произвести сравнение на «меньше». Когда это не подходит, программист может определить свой собственный критерий сравнения, передавая его в качестве параметра. Однако для проверки на равенство такого механизма передачи нет. Вместо этого, когда программист определяет свой критерий cmp, выполняются две проверки на равенство:

```
if(x == y)                        // не делается в случае пользовательских сравнений
if(!cmp(x,y) && !cmp(y,x))        // вот что делается в случае пользовательского критерия cmp
```

Таким образом мы избавляемся от необходимости передавать каждому контейнеру и алгоритму критерий проверки на равенство. Это может показаться расточительным, но стандартная библиотека использует проверки на равенство не часто, к тому же в 50% таких случаев требуется всего лишь одно обращение к ***cmp***().

Также в стандартной библиотеке имеют место случаи проверки отношения эквивалентности с помощью сравнения на меньше, а не сравнения на равенство. Например, ассоциативные контейнеры (§17.4) сравнивают ключи с помощью проверки на эквивалентность !(***cmp***(*x,y*)||***cmp***(*y,x*)). Из этого следует, что эквивалентные ключи не обязаны быть равными. Например, контейнер ***multimap*** (§17.4.2), использующий критерий сравнения без учета регистра, рассматривает строки ***Last***, ***last***, ***lAst***, ***laSt*** и ***lasT*** как эквивалентные, хотя операция == считает их разными. Это позволяет игнорировать при сортировке те различия, которые мы считаем несущественными.

При наличии операций < и == мы можем легко сконструировать остальные операции сравнения. Стандартная библиотека определяет их в пространстве имен ***std***::***rel_ops*** и расположены они в заголовочном файле <***utility***>:

```
template<class T> bool rel_ops::operator!=(const T& x, const T& y) {return !(x==y);}
template<class T> bool rel_ops::operator>(const T& x, const T& y) {return y<x;}
template<class T> bool rel_ops::operator<=(const T& x, const T& y) {return !(y<x);}
template<class T> bool rel_ops::operator>=(const T& x, const T& y) {return !(x<y);}
```

Размещение этих операций в ***rel_ops*** гарантирует их простое использование при необходимости, в то же время уберегая нас от их неявного создания:

```
void f()
{
    using namespace std;
    // !=, >, и т.д.
}

void g()
{
    using namespace std;
    using namespace std::rel_ops;
    // !=, >, и т.д.
}
```

Операции != и др. не определяются в пространстве имен ***std*** потому, что они далеко не всегда нужны, а иногда их определение может вступить в противоречие с кодом пользователя. Например, если бы я писал некоторую общую математическую библиотеку, мне наверняка потребовались бы свои собственные операции сравнения, а не их версии из стандартной библиотеки.

17.2. Последовательные контейнеры

Все последовательные контейнеры следуют модели, описанной для *векторов* (§16.3). Стандартная библиотека содержит следующие фундаментальные последовательные контейнеры:

- ***vector*** (вектор)
- ***list*** (список)
- ***deque*** (двусторонняя очередь)

Менее фундаментальные последовательные контейнеры строятся поверх фундаментальных предоставлением соответствующего интерфейса:

- ***stack*** (стек)
- ***queue*** (очередь)
- ***priority_queue*** (очередь с приоритетом)

Их называют контейнерными адаптерами или просто *адаптерами* (*adapters*) (§17.3).

17.2.1. Контейнер vector

Стандартный контейнер ***vector*** в деталях описан в §16.3. Средства резервирования памяти (§16.3.8) уникальны для этого контейнера. По умолчанию, операция индексирования не проверяет диапазон индексов. Если нужна проверка, пользуйтесь ***at***() (§16.3.3), вектором с проверкой (§3.7.2) или итераторами с проверкой (§19.3). Контейнер ***vector*** предоставляет итераторы с произвольным доступом (§19.2.1).

17.2.2. Контейнер list

Контейнер ***list*** (список) — это последовательный контейнер, *оптимизированный под операции вставки и удаления элементов*. По сравнению с векторами (и контейнерами ***deque***; §17.2.3) операция индексации для списков чрезвычайно медлительна, так что в ***list*** ее даже и не включают. В итоге ***list*** *предоставляет* не итераторы произвольного доступа, а *двунаправленные итераторы* (§19.2.1). Все это обуславливает типичную реализацию ***list*** в виде той или иной формы структур данных, называемых *двусвязными списками* (*doubly-linked list*) (§17.8[16]).

Контейнер ***list*** предоставляет все типы и операции, предоставляемые контейнером ***vector*** (§16.3), за исключением индексации, ***capacity***() и ***reserve***():

```
template<class T, class A = allocator<T> > class std::list
{
public:
```

```
    // типы и операции как у vector, кроме [], at(), capacity() и reserve()
    // ...
};
```

17.2.2.1. Операции splice(), sort() и merge()

Дополнительно к стандартным для всех последовательных контейнеров операциям *list* реализует еще и операции, предназначенные для специфических манипуляций со списками:

```
template<class T, class A = allocator<T> > class list
{
public:
    // ...
    // операции, специфичные для списков:
    void splice (iterator pos, list& x) ;         // перемещение всех эл-ов из x на место перед
                                                  // pos в данном списке без копирования.
    void splice (iterator pos, list& x, iterator p) ; // то же самое, но с позиции p из x.
    void splice (iterator pos, list& x, iterator first, iterator last) ;

    void merge (list&) ;                          // слияние сортированных списков
    template<class Cmp> void merge (list&, Cmp) ;

    void sort () ;
    template<class Cmp> void sort (Cmp) ;
    // ...
};
```

Эти операции характеризуются *устойчивостью* (*stable operations*) в том смысле, что они сохраняют относительный порядок эквивалентных элементов.

«Фруктовый» пример из §16.3.6 переиначим сейчас на контейнер *fruit* типа *list*. Теперь мы можем извлекать элементы из одного списка фруктов и внедрять их в другой список единственной операцией *splice*(). Если имеются два списковых контейнера фруктов

```
fruit:
        apple      pear
citrus:
        orange     grapefruit     lemon
```

мы можем переместить *orange* (апельсин) из списка *citrus* в список *fruit* следующим образом:

```
list<string>::iterator p = find_if(fruit.begin(),fruit.end(),initial('p'));
fruit.splice(p,citrus,citrus.begin());
```

Здесь из *citrus* удаляется первый элемент (указуется итератором *citrus.begin*()), и этот элемент вставляется в список *fruit* перед первых элементом, начинающимся на букву *p*. Вот что из этого получается:

```
fruit:
        apple      orange     pear
citrus:
        grapefruit     lemon
```

Отметим, что операция *splice*() не копирует элементы таким же образом, как это делает *insert*() (§16.3.6). Вместо этого она просто изменяет внутренние указатели на данные, относящиеся к элементам.

Операция *splice*() помимо удаления-вставки индивидуальных элементов и их диапазонов может все это выполнить для целого списка:

```
fruit.splice(fruit.begin(), citrus);
```

Это дает следующий результат:

```
fruit:
        grapefruit    lemon    apple    orange    pear
citrus:
        <empty>
```

В любых версиях функция *splice*() в качестве второго аргумента получает список, из которого элементы забираются. Это позволяет забирать элементы из точно специфицированного источника — тут итератора было бы недостаточно, ибо нет способа выявить контейнер по одному лишь итератору на его элемент (§18.6).

Естественно, аргументы-итераторы должны быть корректны по отношению к спискам, на элементы которых они якобы указывают. То есть они должны либо указывать на действительный элемент списка, либо иметь значение, возвращаемое функцией *end*(). В противном случае результат не определен и потенциально катастрофичен. Например:

```
list<string>::iterator p = find_if(fruit.begin(),fruit.end(),initial('p'));
fruit.splice(p,citrus,citrus.begin()); // ok
fruit.splice(p,citrus,fruit.begin());  // error: fruit.begin() не указывает внутрь citrus
citrus.splice(p,fruit,fruit.begin());  // error: p не указывает внутрь citrus
```

Первый вызов *splice*() корректен даже в случае, когда контейнер *citrus* пуст.

Операция *merge*() (слияние) объединяет два сортированных списка, изымая их из одного списка и вставляя в другой с сохранением порядка. Например, списки

```
f1:
        apple    quince    pear
f2:
        lemon    grapefruit    orange    lime
```

могут быть отсортированы и объединены следующим образом:

```
f1.sort();
f2.sort();
f1.merge(f2);
```

В результате получится

```
f1:
        apple    grapefruit    lemon    lime    orange    pear    quince
f2:
        <empty>
```

Если один из объединяемых списков не отсортирован, функция *merge*() все равно породит объединенный список, однако по поводу его упорядоченности нет никаких гарантий.

Как и операция *splice*(), функция *merge*() фактически не копирует элементы. Вместо этого она удаляет элементы из списка-источника и на уровне внутренних структур данных вставляет их в список-приемник. После выполнения *x.merge(y)*, список y оказывается пустым.

17.2.2.2. «Головные» операции

«Головные» операции, то есть операции над самыми первыми (находящимися в голове) элементами списков, призваны дополнить «концевые» операции, присущие всем последовательным контейнерам (§16.3.6):

```
template <class T, class A = allocator<T> > class list
{
public:
  // ...
  // доступ к элементам:
  reference front();                      // ссылка на первый элемент
  const_reference front() const;

  void push_front(const T&);              // добавить новый первый элемент
  void pop_front();                       // удалить первый элемент
  // ...
};
```

Головной (самый первый) элемент контейнера принято называть *front*. Для списков *головные операции* (*front operations*) столь же эффективны и удобны, как и *концевые операции* (*back operations*) (§16.3.5). Если имеется возможность выбора, то лучше предпочесть концевые операции головным операциям, поскольку код, использующий лишь концевые операции, одинаково хорошо подходит и для списков, и для векторов. То есть если существует ненулевая вероятность того, что код, изначально написанный для списков, постепенно может эволюционировать в обобщенный код, пригодный для всех типов контейнеров, лучше ограничиться более широко распространенными концевыми операциями. Это частный случай правила, гласящего, что для того, чтобы добиться максимальной гибкости, нужно применять минимальный набор операций (§17.1.4.1).

17.2.2.3. Другие операции

Вставка и удаление элементов особо эффективны для *списков*. Это заставляет людей предпочесть контейнер *list* в случаях, когда приходится выполнять их часто. В свою очередь, это требует прямой реализации методов удаления элементов:

```
template <class T, class A = allocator<T> > class list
{
public:
  // ...
  void remove(const T& val);
  template<class Pred> void remove_if(Pred p);

  void unique();                                  // удаляет дубликаты с помощью ==
  template<class BinPred> void unique(BinPred b); // удаляет дубликаты, используя b;
  void reverse();                                 // обратный порядок элементов
};
```

Например, для контейнера

```
fruit:
    apple  orange  grapefruit  lemon  orange  lime  pear  quince
```

мы можем следующим образом удалить все элементы со значением ***orange***:

```
fruit.remove("orange");
```

При этом получится следующий результат:

```
fruit:
    apple  grapefruit  lemon  lime  pear  quince
```

Часто при удалении элементов более интересны иные критерии, чем просто удаление элемента с заданным значением. Операция ***remove_if***() воспринимает критерии удаления. Например,

```
fruit.remove_if(initial('l'));
```

удаляет элементы, начинающиеся на букву ***l***, что приводит к следующему результату:

```
fruit:
    apple  grapefruit  pear  quince
```

Удаление элементов часто производится с целью устранения дубликатов. Для этого специально предназначена функция ***unique***():

```
fruit.sort();
fruit.unique();
```

Причина предварительной сортировки заключается в том, что ***unique***() удаляет лишь смежные дубликаты. Например, будь содержимое списка следующим

```
apple  pear  apple  apple  pear
```

простой вызов ***fruit.unique***() приведет к

```
apple  pear  apple  pear
```

а в случае предварительной сортировке — к

```
apple  pear
```

Если нужно удалить лишь определенные дубликаты, мы можем предоставить предикат (условие), специфицирующий, какие именно дубликаты должны быть удалены. Например, можно определить бинарный предикат (*binary predicate*, §18.4.2) ***initial2***(***x***), чтобы выявлять строки, начинающиеся на букву ***x***, и возвращающий ***false*** в иных случаях (когда строка не начинается на букву ***x***). Например, если в списке содержатся фрукты

```
pear  pear  apple  apple
```

мы можем удалить последовательные дубликаты фруктов, начинающихся на букву ***p***, вызовом

```
fruit.unique(initial2('p'));
```

Это даст следующий список фруктов:

```
pear  apple  apple
```

Как отмечено в §16.3.2, иногда требуется просматривать контейнер в обратном порядке. Для списков имеется возможность инвертировать порядок элементов без необходимости их копирования. Это выполняется функцией ***reverse***(). Пусть задан следующий список:

fruit:
banana ***cherry*** ***lime*** ***strawberry***

Тогда вызов ***fruit.reverse***() даст следующий результат:

fruit:
strawberry ***lime*** ***cherry*** ***banana***

Удаляемый из списка элемент уничтожается. Однако заметьте, что уничтожение указателя не есть уничтожение указуемого объекта. Если вам нужен контейнер указателей, который при удалении из него элемента (то есть указателя) уничтожает указуемый объект, вам придется написать его самому (§17.8[13]).

17.2.3. Контейнер deque

Контейнер ***deque*** реализует структуру данных, называемую *двусторонней очередью* (*double-ended queue*). То есть это последовательный контейнер, оптимизированный таким образом, что головные и концевые операции для него почти так же эффективны, как для *списков*, а индексирование почти так же эффективно, как для векторов:

```
template<class T, class A = allocator<T> > class std::deque
{
  // типы и операции как у vector (§16.3.3, §16.3.5, §16.3.6)
  // кроме capacity() и reserve()
  // плюс "головные" операции (§17.2.2.2) как у list
};
```

В то же время, вставка и удаление элементов где-нибудь в «середине контейнера» неэффективны так же, как и для векторов. Следовательно, ***deque*** используется в случаях, когда добавление элементов и их удаление производятся на обоих краях (в начале или конце) контейнера. Например, можно применить двустороннюю очередь для моделирования участка железной дороги или для представления колоды карт в карточной игре:

```
deque<car> siding_no_3;
deque<Card> bonus;
```

17.3. Адаптеры последовательных контейнеров

Последовательные контейнеры ***vector***, ***list*** и ***deque*** нельзя построить один из другого без потери эффективности. С другой стороны, можно элегантно и эффективно реализовать стеки и очереди при помощи этих трех фундаментальных последовательных контейнеров. Поэтому ***stack*** и ***queue*** определяются не как независимые контейнеры, а как адаптеры фундаментальных последовательных контейнеров.

Контейнерный адаптер (*container adapter*) призван предоставлять *ограниченный интерфейс к контейнеру*. В частности, адаптеры не предоставляют итераторов, так как предполагается, что адаптерные контейнеры нужно использовать только через их специализированный интерфейс.

Методы, с помощью которых из фундаментальных контейнеров создаются адаптерные контейнеры, могут в общем случае применяться при неинтрузивном подстраивании классового интерфейса под нужды пользователей.

17.3.1. Стек

Адаптер **stack** определен в заголовочном файле <**stack**>. Он столь прост, что для его изучения лучше всего просто представить его реализацию:

```
template<class T, class C = deque<T> > class std::stack
{
protected:
  C c;

public:
  typedef typename C::value_type value_type;
  typedef typename C::size_type size_type;
  typedef C container_type;

  explicit stack(const C& a = C()) : c(a) {}

  bool empty() const {return c.empty();}
  size_type size() const {return c.size();}

  value_type& top() {return c.back();}
  const value_type& top() const {return c.back();}

  void push(const value_type& x) {c.push_back(x);}
  void pop() {c.pop_back();}
};
```

Таким образом, **stack** есть просто интерфейс для контейнера, который передается ему в качестве шаблонного параметра. Все, что делает стек, сводится к удалению из интерфейса нестековых операций и к переименовыванию операций **back**(), **push_back**() и **pop_back**() в более традиционные для стеков имена — **top**(), **push**() и **pop**().

По умолчанию стек хранит свои элементы в двусторонней очереди, но можно использовать и любой другой последовательный контейнер, предоставляющий операции **back**(), **push_back**() и **pop_back**(). Например:

```
stack<char> s1;                 // используем deque<char> для хранения элементов типа char
stack<int,vector<int> > s2;     // используем vector<int> для хранения элементов типа int
```

Можно предоставить существующий контейнер для инициализации стека:

```
void print_backwards(vector<int>& v)
{
  stack<int,vector<int> > state(v);      // инициализируем state из v
  while(state.size())
  {
```

```
    cout<< state.top();
    state.pop();
  }
}
```

Надо иметь в виду, что элементы аргумента-контейнера копируются, так что предоставление существующего контейнера в качестве аргумента функции довольно дорогостоящая затея.

Элементы добавляются в стек при помощи операции ***push_back***() базового контейнера, используемого для хранения элементов. Следовательно, стек не может переполниться, пока имеется достаточно памяти для выделения под элементы этого контейнера (с помощью его же аллокатора; см. §19.4).

С другой стороны, стеку может грозить «переисчерпание» (underflow):

```
void f()
{
  stack<int, vector<int> > s;
  s.push(2);
  if(s.empty())                    // защита от underflow
  {
    // не делать pop
  }
  else
  {
    s.pop();                       // чудесно: s.size() становится 0
    s.pop();                       // эффект не определен, возможны проблемы
  }
}
```

Отметим, что для использования элемента стека нет необходимости выполнять операцию pop(). Вместо этого используют ***top***(), а ***pop***() применяют тогда, когда элемент больше не нужен. Это не вызывает особых затруднений и более эффективно в случаях, когда нет нужды в операции ***pop***():

```
void f(stack<char>& s)
{
  if(s.top()=='c') s.pop();        // удаляем возможные начальные 'c'
  // ...
}
```

В отличие от фундаментальных контейнеров ***stack*** (как и другие адаптеры) не имеет среди параметров шаблона распределителя памяти. Вместо этого он полагается на распределитель памяти базового фундаментального контейнера.

17.3.2. Очередь

Адаптер ***queue***, определенный в заголовочном файле <***queue***>, реализует интерфейс к контейнеру, позволяющему *добавлять элементы в конец и извлекать их с головы* (структуры данных с такими свойствами называют *очередями* — *queues*):

```
template<class T, class C = deque<T> > class std::queue
{
```

```
protected:
  C c;

public:
  typedef typename C::value_type value_type;
  typedef typename C::size_type size_type;
  typedef C container_type;

  explicit queue(const C& a=C()) : c(a) {}

  bool empty() const {return c.empty();}
  size_type size() const {return c.size();}

  value_type& front() {return c.front();}
  const value_type& front() const {return c.front();}

  value_type& back() {return c.back();}
  const value_type& back() const {return c.back();}

  void push(const value_type& x) {c.push_back(x);}
  void pop() {c.pop_front();}
};
```

По умолчанию очередь ***queue*** использует для хранения элементов контейнер ***deque***, но можно использовать и любой другой последовательный контейнер, предоставляющий операции ***front***(), ***back***(), ***push_back***() и ***pop_front***(). Поскольку ***vector*** не предоставляет операции ***pop_front***(), он не может использоваться в качестве базового фундаментального контейнера для хранения элементов очереди.

Очереди находят применение практически во всех системах. Например, для системы передачи и обработки сообщений можно определить следующий сервер:

```
struct Message
{
  // ...
};

void server(queue<Message>& q)
{
  while(!q.empty())
  {
    Message& m = q.front();     // получить сообщение
    m.service();                // обслужить запрос
    q.pop();                    // уничтожить сообщение
  }
}
```

Сообщения помещаются в очередь операцией ***push***().

Если сервер и клиентская программа работают в разных процессах или потоках, потребуется некоторое средство синхронизации доступа к очереди. Например:

```
void server2(queue<Message>& q, Lock& lck)
{
  while(!q.empty())
  {
    Message m;
```

```
    {
      LockPtr h(lck);           // блокировка на время получения сообщения (см. §14.4.1)
      if(q.empty()) return;     // кто-то другой получил сообщение
      m=q.front();
      q.pop();
    }
    m.service();
  }
}
```

В языке С++ нет стандартных средств реализации параллельной работы и взаимоблокировок. Посмотрите, что может предложить ваша система и как это реализовать на С++ (§17.8[8]).

17.3.3. Очередь с приоритетом

Очередь с приоритетом (***priority_queue***) — это такая очередь, которая каждому элементу назначает приоритет, определяющий порядок, в котором элементы просматриваются (операция ***top***()) пользователем:

```
template<class T, class C = vector<T>, class Cmp=less<typename C::value_type> >
class std::priority_queue
{
protected:
  C c;
  Cmp cmp;

public:
  typedef typename C::value_type value_type;
  typedef typename C::size_type size_type;
  typedef C container_type;

  explicit priority_queue(const Cmp& a1 = Cmp(), const C& a2 = C())
        : c(a2), cmp(a1) {make_heap(c.begin(),c.end(),cmp);} // см. §18.8
  template<class In>
  priority_queue(In first, In last, const Cmp& = Cmp(), const C& = C());

  bool empty() const {return c.empty();}
  size_type size() const {return c.size();}

  const value_type& top() const {return c.front();}

  void push(const value_type&);
  void pop();
};
```

Объявление ***priority_queue*** находится в заголовочном файле <***queue***>.

По умолчанию, ***priority_queue*** упорядочивает элементы, сравнивая их с помощью операции <, а операция ***top***() возвращает наибольший элемент:

```
struct Message
{
  int priority;
  bool operator<(const Message& x) const {return priority<x.priority;}
```

```
  // ...
};

void server(priority_queue<Message>& q, Lock& lck)
{
  while(!q.empty())
  {
    Message m;
    {
      LockPtr h(lck);             // блокировка на время получения сообщения (см. §14.4.1)
      if(q.empty()) return;       // кто-то другой получил сообщение
      m = q.top();
      q.pop();
    }
    m.service();
  }
}
```

Этот пример отличается от примера с очередью (§17.3.2) тем, что сообщения с более высоким приоритетом будут обслуживаться первыми. Порядок, в котором элементы с одинаковым приоритетом поступают в голову ***priority_queue***, не определен. Два элемента считаются одинаково приоритетными, если ни у одного из них приоритет не выше, чем у другого (§17.4.1.5).

Альтернативный операции < критерий сравнения элементов можно передать в виде аргумента шаблона. Например, можно отсортировать строки без учета регистра букв, поместив их в

```
priority_queue<string,vector<string>,Nocase> pq;    // Nocase для сравнений (§17.1.4.1)
```

при помощи ***pq.push()*** и извлекая их при помощи ***pq.top()*** и ***pq.pop()***.

Отметим, что объекты, создаваемые из шаблонов, которым передаются разные шаблонные аргументы, имеют разный тип (§13.6.3.1):

```
priority_queue<string>& pq1 = pq;    // error: несовпадение типов
```

Но мы можем предоставить критерий сравнения, не затрагивая типа ***priority_queue***, просто передав его конструктору в виде аргумента:

```
struct String_cmp    // тип, представляющий критерий сравнения на этапе выполнения
{
  String_cmp(int n = 0);
  // ...
};

typedef priority_queue<string, vector<string>, String_cmp> Pqueue;

void g(Pqueue& pq)                      // pq использует String_cmp() для сравнений
{
  Pqueue pq2(String_cmp(nocase));       // "nocase" - сравнение без учета регистра букв
  pq = pq2;                             // ok: pq и pq2 - одного типа,
                                        // pq теперь также использует String_cmp(nocase)
}
```

Поддержание упорядоченного хранения элементов не дается задаром, но оно и не столь дорого. В одной из реализаций ***priority_queue*** с целью отслеживания от-

носительного положения элементов используется дерево, что для операций ***push***() и ***pop***() приводит к оценке ***O***(***log***(***n***)).

По умолчанию очередь с приоритетом использует для хранения своих элементов вектор, но можно использовать и любой другой последовательный контейнер, предоставляющий операции ***front***(), ***push_back***(), ***pop_back***() и итераторы произвольного доступа. Чаще всего реализации ***priority_queue*** так или иначе используют ***heap*** (§18.8).

17.4. Ассоциативные контейнеры

Ассоциативный массив (*associative array*) — один из самых полезных универсальных пользовательских типов. Более того, в языках, ориентированных на обработку текстов и символов, это зачастую встроенный тип. Ассоциативный массив, часто называемый также *отображением* (*map*) или *словарем* (*dictionary*), содержит пары значений. Зная одно значение, называемое *ключом* (*key*), можно получить доступ к другому значению, называемому *отображенным значением* (*mapped value*). Ассоциативный массив можно представлять себе как вектор, у которого индекс не обязательно целочисленный:

```
template<class K, class V> class Assoc
{
public:
  V& operator[] (const K&) ;    // возвращает ссылку на V, соответствующий K
  // ...
};
```

Таким образом по ключу типа ***K*** находится отображенное значение типа ***V***.

Ассоциативные контейнеры являются обобщением понятия ассоциативного массива. Например, контейнер ***map*** — это традиционный ассоциативный массив, в котором единственное значение ассоциируется с уникальным ключом. Ассоциативный контейнер ***multimap*** является ассоциативным массивом, для которого каждому ключу может соответствовать несколько значений; ***set*** и ***multiset*** являются вырожденными ассоциативными массивами, для которых ключу не ставится в соответствие никаких значений.

17.4.1. Ассоциативный массив map

Ассоциативный массив ***map*** — это *последовательность пар* (*ключ,значение*), которая обеспечивает *быстрое нахождение значения по ключу*. Каждому ключу соответствует максимум одно значение. Иными словами каждый ключ уникален. Контейнер map предоставляет двунаправленные итераторы (§19.2.1).

Контейнер ***map*** требует, чтобы для типа ключа существовала операция < (§17.1.4.1); он хранит элементы отсортированными, так что перебор элементов осуществляется упорядоченным образом. Для элементов, не имеющих очевидного критерия упорядочения, или в случаях, когда не требуется хранить элементы упорядоченным образом, можно предложить контейнер ***hash_map*** (§17.6).

17.4.1.1. Типы

Ассоциативный массив ***map*** определяет традиционные типы (§16.3.1) плюс ряд типов, отражающих его специфику:

```
template<class Key, class T, class Cmp=less<Key>, class A=allocator<pair<const Key, T> > >
class std::map
{
public:
  // типы:
  typedef Key key_type;
  typedef T mapped_type;

  typedef pair<const Key, T> value_type;

  typedef Cmp key_compare;
  typedef A allocator_type;

  typedef typename A::reference reference;
  typedef typename A::const_reference const_reference;

  typedef implementation_defined1 iterator;
  typedef implementation_defined2 const_iterator;

  typedef typename A::size_type size_type;
  typedef typename A::difference_type difference_type;

  typedef std::reverse_iterator<iterator> reverse_iterator;
  typedef std::reverse_iterator<const_iterator> const_reverse_iterator;
  // ...
};
```

Обратите внимание на то, что ***value_type*** контейнера ***map*** это пара (ключ,значение). Тип отображенных значений обозначен как ***mapped_type***. Таким образом, map есть последовательность элементов типа ***pair<const Key, mapped_type>***.

Как обычно, фактические типы итераторов определяются конкретной реализацией. Поскольку map, вероятнее всего, реализуется в виде дерева, то его итераторы призваны обеспечить проход по всем узлам дерева.

Обратные итераторы конструируются из стандартных шаблонов ***reverse_iterator*** (§19.2.5).

17.4.1.2. Итераторы

Контейнер ***map*** обеспечивает обычный набор функций, возвращающих итераторы (§16.3.2):

```
template<class Key, class T, class Cmp = less<Key>,
         class A = allocator<pair<const Key, T> > >
class std::map
{
public:
  // ...
  // итераторы:
  iterator begin();
  const_iterator begin() const;

  iterator end();
  const_iterator end() const;
```

```
    reverse_iterator rbegin () ;
    const_reverse_iterator rbegin () const;

    reverse_iterator rend () ;
    const_reverse_iterator rend () const;
    // ...
};
```

Итерации по контейнеру ***map*** есть итерации по последовательности элементов типа ***pair<const Key, mapped_type>***. Например, можно распечатать записи телефонной книги следующим образом:

```
void f(map<string, number>& phone_book)
{
    typedef map<string, number>: :const_iterator CI;

    for (CI p = phone_book.begin () ; p != phone_book.end () ; ++p)
        cout<< p->first << '\t' << p->second << '\n';
}
```

Итераторы ассоциативных массивов представляют элементы в порядке возрастания их ключей (§17.4.1.5), и поэтому телефонная книга ***phone_book*** будет распечатана в лексикографическом порядке.

Мы обращаемся к первому элементу любой пары по идентификатору ***first***, а ко второму — по идентификатору ***second*** независимо от их фактических типов:

```
template<class T1, class T2> struct std: :pair
{
    typedef T1 first_type;
    typedef T2 second_type;

    T1 first;
    T2 second;

    pair () : first (T1 () ) , second (T2 () ) { }
    pair (const T1& x, const T2& y) : first (x) , second (y) { }
    template<class U, class V>
    pair (const pair<U, V>& p) : first (p.first) , second (p.second) { }
};
```

Последний конструктор предназначен для преобразования пар (§13.6.2). Например:

```
pair<int, double> f(char c, int i)
{
    pair<char, int> (c, i) ;
    // ...
    return x;      // преобразование pair<char,int> в pair<int,double>
}
```

В контейнере ***map*** ключ является первым элементом в паре, а отображенное значение — вторым.

Полезность типа ***pair*** не ограничивается контейнером ***map***; он является полноправным классом стандартной библиотеки языка C++. Определение ***pair*** расположено в заголовочном файле <***utility***>. Также имеется функция для удобного составления пар:

```
template<class T1, class T2> pair<T1,T2> std::make_pair(T1 t1, T2 t2)
{
  return pair<T1,T2>(t1,t2);
}
```

По умолчанию пара инициализируется умолчательными значениями типов ее элементов. Это приводит к тому, что элементы встроенных типов инициализируются нулями (§5.1.1), а элементы типа ***string*** инициализируются пустой строкой (§20.3.4). Для типов без умолчательных конструкторов создание пар возможно лишь в форме явной инициализации.

17.4.1.3. Индексация

Для контейнера ***map*** характерной операцией является поиск по ключу в рамках операции индексации:

```
template<class Key, class T, class Cmp = less<Key>,
         class A=allocator<pair<const Key,T> > >
class map
{
public:
  // ...
  mapped_type& operator[] (const key_type& k);    // доступ к элементу с ключом k
  // ...
};
```

Операция индексации выполняет поиск по ключу, заданному в качестве индекса, и возвращает соответствующее отображенное значение. Если заданный в операции индексации ключ в контейнере не обнаруживается, элемент с этим ключом и умолчательным отображенным значением типа ***mapped_type*** внедряется в контейнер ***map***. Например:

```
void f()
{
  map<string,ing> m;  // пустой массив типа map
  int x = m["Henry"]; // создается новый вход для "Henry", инициал-ся 0, возвращает 0

  m["Harry"] = 7;     // создается новый вход для "Harry", инициал-ся 0, присваивает 7

  int y = m["Henry"]; // возвращает значение для входа "Henry"
  m["Harry"] = 9;     // заменяем значение для входа "Harry" на 9
}
```

В качестве более реалистичного примера рассмотрим задачу вычисления вырученных сумм для всех предметов, представленных в виде пар (имя предмета,значение):

```
nail 100 hammer 2 saw 3 saw 4 hammer 7 nail 1000 nail 250
```

а также сумм по каждому предмету. Основную работу можно выполнить в процессе считывания пар (имя предмета,значение) в контейнер ***m*** типа ***map***:

```
void readitems(map<string,int>& m)
{
  string word;
```

```
    int val = 0;
    while (cin >> word >> val) m[word] += val;
}
```

Операция индексирования *m*[*word*] определяет соответствующую пару (*string*, *int*) и возвращает ссылку на ее целочисленную часть. Этот код пользуется тем фактом, что в новом элементе целая часть по умолчанию устанавливается равной нулю.

Созданный функцией *readitems*() можно легко распечатать при помощи традиционного цикла:

```
int main ()
{
    map<string, int> tbl;
    readitems (tbl) ;

    int total = 0;
    typedef map<string, int>::const_iterator CI;

    for (CI p = tbl.begin () ; p != tbl.end () ; ++p)
    {
        total += p->second;
        cout<< p->first << '\t' << p->second << '\n';
    }

    cout<< "----------------\ntotal \t" << total<< '\n';
    return !cin;
}
```

что даст следующий результат:

```
hammer  9
nail    1350
saw     7
----------------
Total   1366
```

Обратите внимание на то, что предметы распечатаны в лексикографическом порядке (§17.4.1, §17.4.1.5).

Операция индексирования должна найти ключ в ассоциативном массиве типа map. Это, конечно, не так дешево, как индексация массива целым числом. Цена равна *O*(*log*(*size_of_map*)), что приемлемо для многих приложений. А для тех из них, где это неприемлемо, выходом может оказаться хэшированный контейнер (§17.6).

Когда ключ не находится, операция индексации добавляет элемент с умолчательным значением. Поэтому для константных ассоциативных массивов не существует соответствующей версии операции индексации. И, вообще, операция индексации выполнима только если *mapped_type* имеет значение по умолчанию. Если программист хочет просто посмотреть, имеется ли уже данный ключ в контейнере, он может применить операцию *find*() (§17.4.1.6), которая не изменяет ассоциативный массив *map*.

17.4.1.4. Конструкторы

Контейнер ***map*** предоставляет обычный набор конструкторов и прочих операций (§16.3.4):

```
template<class Key, class T, class Cmp=less<Key>, class A=allocator<pair<const Key, T> > >
class map
{
public:
  // ...
  // конструирование/копирование/уничтожение:
  explicit map (const Cmp& c=Cmp (), const A& = A ());
  template<class In> map (In first, In last, const Cmp& c=Cmp (), const A& = A ());
  map (const map&);

  ~map ();

  map& operator= (const map&);
  // ...
};
```

Копирование контейнера предполагает выделение памяти под элементы и копирование каждого из них (§16.3.4). Все это весьма дорого, так что пользоваться этим нужно лишь при необходимости. Как следствие, контейнеры типа ***map*** передают, чаще всего, по ссылке.

Шаблонный конструктор принимает диапазон пар ***pair<const Key, T>***, задаваемый входными итераторами. При помощи функции ***insert*** () (§17.4.1.7) он вставляет элементы этого диапазона в контейнер ***map***.

17.4.1.5. Сравнения

Чтобы найти элемент в ассоциативном массиве по заданному ключу, операции этого контейнера должны сравнивать ключи. Итераторы обеспечивают перемещение по ассоциативному массиву в порядке возрастания значений ключей, так что при вставке элемента ключи тоже будут сравниваться (перед вставкой элемента в древовидную структуру, представляющую контейнер ***map***).

По умолчанию, для сравнения ключей используется операция <, но можно предоставить и альтернативу в виде параметра шаблона или аргумента конструктора (§17.3.3). Это именно сравнение ключей, в то время как ***value_type*** для ***map*** есть пара (ключ,значение). Поэтому функция ***value_comp*** () определяется таким образом, чтобы сравнивать пары, используя функцию сравнения ключей:

```
template<class Key, class T, class Cmp = less<Key>,
         class A = allocator<pair<const Key, T> > >
class map
{
public:
  // ...
  typedef Cmp key_compare;

  class value_compare: public binary_function<value_type, value_type, bool>
  {
    friend class map;
```

```
  protected:
    Cmp cmp;
    value_compare (Cmp c) : cmp (c) { }
  public:
    bool operator () (const value_type& x, const value_type& y) const
    { return cmp (x.first, y.first) ; }
  };

  key_compare key_comp () const;
  value_compare value_comp () const;
  // ...
};
```

Например:

```
map<string, int> m1;
map<string, int, Nocase> m2;                    // задаем тип сравнения (§17.1.4.1)
map<string, int, String_cmp> m3;                // задаем тип сравнения (§17.1.4.1)
map<string, int, String_cmp> m4 (String_cmp (literary) ) ; // передаем объект сравнения
```

Функции-члены ***key_comp***() и ***value_comp***() позволяют запрашивать ассоциативные массивы о способах, которыми они сравнивают ключи и значения. Обычно это делается с целью обеспечить тот же самый критерий сравнения для других контейнеров и алгоритмов. Например:

```
void f(map<string, int>& m)
{
  map<string, int> m2;              // сравнение с помощью < (умолчательный вариант)
  map<string, int> m3 (m.key_comp () ) ;  // с равнение с помощью m
  // ...
}
```

О том, как определить специфическое сравнение, рассказано в §17.1.4.1; общее описание классов функциональных объектов дано в §18.4.

17.4.1.6. Специфические для контейнера map операции

Главная идея ассоциативных массивов (и вообще всех ассоциативных контейнеров) состоит в получении информации по ключу. Для этого в контейнер ***map*** введено несколько специфических операций:

```
template<class Key, class T, class Cmp = less<Key>,
         class A = allocator<pair<const Key, T> > >
class map
{
public:
  // ...
  // операции ассоциативных массивов:
  iterator find (const key_type& k) ;           // находит элемент с ключом k
  const_iterator find (const key_type& k) const;

  size_type count (const key_type& k) const;    // находит число элементов с ключом k
```

```
iterator lower_bound(const key_type& k);      // находит первый элемент с ключом k
  const_iterator lower_bound(const key_type& k) const;

  iterator upper_bound(const key_type& k); // находит первый эл-т с ключом, большим k
  const_iterator upper_bound(const key_type& k) const;

  pair<iterator,iterator> equal_range(const key_type& k);
  pair<const_iterator,const_iterator> equal_range(const key_type& k) const;
  // ...
};
```

Выражение ***m.find(k)*** просто выдает итератор, соответствующий элементу с ключом ***k***. Если такого элемента не существует, возвращается итератор ***m.end()***. Для контейнеров с уникальными ключами, таких как ***map*** или ***set***, результирующий итератор будет указывать на уникальный элемент с заданным ключом ***k***. Для контейнеров, не имеющих уникальных ключей, таких как ***multimap*** или ***multiset***, результирующий итератор будет указывать на первый элемент с таким ключом. Например:

```
void f(map<string,int>& m)
{
  map<string,int>::iterator p = m.find("Gold");
  if(p!=m.end())                      // если найден "Gold"
  {
    // ...
  }
  else if(m.find("Silver")!=m.end())   // ищем "Silver"
  {
    // ...
  }
  // ...
}
```

Для контейнера ***multimap*** (§17.4.2) нахождение первого вхождения редко когда полезно; выражения ***m.lower_bound(k)*** и ***m.upper_bound(k)*** дают начало и конец подпоследовательности элементов контейнера ***m*** с заданным ключом ***k***. Как обычно, конец подпоследовательности фиксируется итератором, указывающим на элемент, расположенный за последним элементом подпоследовательности. Например:

```
void f(multimap<string,int>& m)
{
  multimap<string,int>::iterator lb = m.lower_bound("Gold");
  multimap<string,int>::iterator ub = m.upper_bound("Gold");

  for (multimap<string,int>::iterator p = lb; p!=ub; ++p)
  {
    // ...
  }
}
```

Нахождение верхней и нижней границ двумя раздельными операциями и не элегантно, и не эффективно. Операция ***equal_range()*** предоставляет обе границы. Например:

```
void f(multimap<string,int>& m)
{
  typedef multimap<string,int>::iterator MI;
  pair<MI,MI> g = m.equal_range("Gold");

  for(MI p = g.first; p!=g.second; ++p)
  {
    // ...
  }
}
```

Если ***lower_bound**(**k**)* не находит ключа ***k***, то она возвращает итератор, настроенный на первый элемент, имеющий ключ, больший ***k***, или возвращает ***end***() в отсутствие такого элемента. Этот способ сигнализации о неудаче применяется также и в функциях ***upper_bound***() и ***equal_range***().

17.4.1.7. Операции, характерные для списков

Стандартным способом введения значения в ассоциативный массив является простое присваивание с применением индексации. Например:

```
phone_book["Order department"] = 8226339;
```

Это гарантирует, что "Отдел заказов" ("Order department") будет внесен в ***phone_book*** (телефонная книга) независимо от того, была ли об этом отделе внесена какая-либо запись ранее. Однако имеется и возможность непосредственно внедрять элементы традиционной для списков функцией ***insert***() и удалять их функцией ***erase***():

```
template<class Key, class T, class Cmp = less<Key>,
     class A = allocator<pair<const Key,T> > >
class map
{
public:
  // ...
  // списковые операции:
  pair<iterator,bool> insert(const value_type& val);   // вставить пару (key,value)
  iterator insert(iterator pos, const value_type& val); // pos - подсказка начала поиска
  template <class In> void insert(In first, In last);   // вставить эл-ы из последоват-ти

  void erase(iterator pos);                   // удалить указуемый элемент
  size_type erase(const key_type& k);         // удалить элемент с ключом k (если он есть)
  void erase(iterator first, iterator last);  // удалить диапазон

  void clear();                               // удалить все элементы
  // ...
};
```

Выражение ***m.insert***(***val***) пытается добавить в ***m*** пару ***val*** типа (***Key***, ***T***). Поскольку ассоциативные массивы ***map*** — это контейнеры с уникальными ключами, такая вставка проходит лишь в случае, когда в ***m*** нет элемента с указанным ключом. Возвращаемое при этом значение — это пара ***pair<iterator,bool>***. Логический элемент пары в случае успеха операции равен ***true***, а итератор указывает на элемент контейнера ***m***, ключ которого равен ***val.first***. Например:

```
void f(map<string,int>& m)
{
  pair<string,int> p99("Paul",99);
  pair<map<string,int>::iterator,bool> p = m.insert(p99);

  if(p.second)
  {
    // "Paul" - вставлен
  }
  else
  {
    // "Paul" уже присутствует
  }

  map<string,int>::iterator i = p.first;     // указывает на m["Paul"]
  // ...
}
```

Обычно, нас при этом не очень заботит, был ли ключ вставлен впервые или уже присутствовал в контейнере типа ***map*** до выполнения операции ***insert***(). Это может быть интересно лишь в связи с необходимостью зафиксировать факт неожиданного его там появления (по внешним для нас причинам). Две другие версии функции ***insert***() не возвращают никаких признаков успешности выполнения операции вставки.

В вызове ***insert***(***pos***, ***val***) позиция ***pos*** — это просто попытка подсказать реализации, где нужно начать поиск ключа ***val.first***. Если подсказка хороша, то можно значительно выиграть в производительности. Если нет, то лучше бы ее вообще не применять (и с точки зрения эффективности, и с точки зрения чистоты записи). Например:

```
void f(map<string,int>& m)
{
  m["Dogbert"] = 3;              // лаконично и четко, но возможно менее эффективно
  m.insert(m.begin(),make_pair(const string("Dogbert"),99));      // уродливо
}
```

Фактически, операция [] — это не просто более удобное обозначение для ***insert***(). Результат выражения ***m***[***k***] эквивалентен результату выражения (*(***m.insert***(***make_pair***(***k***, ***V***())).***first***)).***second***, где ***V***() есть умолчательное значение для отображенного типа. Поняв эту эквивалентность, вы поймете суть ассоциативных контейнеров.

Поскольку операция [] всегда использует ***V***(), нельзя применять индексацию для ассоциативных массивов с отображенными значениями типов, не имеющими умолчательных значений. Это неприятное *ограничение* для всех *стандартных* ассоциативных контейнеров (оно не является фундаментальным ограничением ассоциативных контейнеров вообще; см. §17.6.2).

Элементы с заданным ключом можно удалять. Например:

```
void f(map<string,int>& m)
{
  int count = m erase("Ratbert");
  // ...
}
```

Возвращаемое значение типа ***int*** означает число удаленных элементов. В частности, здесь число ***0*** означает, что нет подлежащего удалению элемента с ключом "***Ratbert***". Для ассоциативных контейнеров типа ***multimap*** и ***multiset*** возвращаемое значение может быть больше ***единицы***. Также имеется возможность удалить элемент, указуемый итератором, или диапазон элементов, начало и конец которого фиксируются двумя итераторами. Например:

```
void g(map<string,int>& m)
{
  m.erase(m.find("Catbert"));
  m.erase(m.find("Alice"),m.find("Wally"));
}
```

Естественно, что быстрее удаляется элемент, непосредственно указуемый итератором, поскольку иначе сначала еще нужно найти его по указанному ключу, и лишь затем удалить. После выполнения операции ***erase***() итератор становится недействительным, ибо элемент, на который он указывал, больше не существует. Вызов ***m.erase(b,e)***, где ***e*** есть ***m.end***(), безопасен (при условии, что ***b*** ссылается на действительный элемент контейнера ***m***, или на ***m.end***()). С другой стороны, вызов ***m.erase(p)***, где ***p*** есть ***m.end***(), является серьезной ошибкой, способной испортить контейнер.

17.4.1.8. Другие функции

Наконец, ассоциативный массив ***map*** предоставляет и традиционные функции, имеющие дело с числом элементов, а также функцию ***swap***():

```
template<class Key, class T, class Cmp = less<Key>,
         class A = allocator<pair<const Key,T> > >
class map
{
public:
  //...
  // емкость:
  size_type size() const;            // число элементов
  size_type max_size() const;        // размер максимально возможного map
  bool empty() const {return size()==0;}

  void swap(map&);
};
```

Как обычно, возвращаемые функциями ***size***() и ***max_size***() значения есть число элементов.

Для контейнера map операции ==, !=, <, >, <=, >= и ***swap***() представлены глобальными шаблонами (а не функциями-членами):

```
template<class Key, class T, class Cmp, class A>
bool operator==(const map<Key,T,Cmp,A>&, const map<Key,T,Cmp,A>&);

// аналогично !=, <,>, <=, и >=

template<class Key, class T, class Cmp, class A>
void swap(map<Key,T,Cmp,A>&, map<Key,T,Cmp,A>&);
```

Зачем может потребоваться сравнивать два контейнера типа ***map***? Ведь когда мы специально сравниваем два ассоциативных массива, мы не интересуемся фактом их различия, а тем, как именно они отличаются. И тем не менее, обеспечив каждый контейнер операциями ==, < и ***swap***(), мы сделаем возможным написание алгоритмов, применимых к любому контейнеру. Например, эти функции позволяют сортировать функцией ***sort***() вектор элементов типа map, а также позволяют реализовать контейнер ***set*** с элементами типа ***map***.

17.4.2. Ассоциативный контейнер multimap

Контейнер ***multimap*** похож на ***map***, но он допускает дублирование ключей:

```
template<class Key, class T, class Cmp = less<Key>,
          class A = allocator<pair<const Key,T> > >
class std::multimap
{
public:
  // как map, за исключением:
  iterator insert(const value_type&);    // возвращает iterator, а не pair
  // нет операции индексирования []
};
```

Например (для сравнения С-строк используем ***Cstring_less*** из §17.1.4.1):

```
void f(map<char*,int,Cstring_less>& m, multimap<char*,int,Cstring_less>& mm)
{
  m.insert(make_pair("x",4));
  m.insert(make_pair("x",5));    // эффекта нет: уже есть вход для "x" (§17.4.1.7)
  // здесь m["x"] == 4

  mm.insert(make_pair("x",4));
  mm.insert(make_pair("x",5));
  // mm содержит как ("x",4), так и ("x",5)
}
```

Это означает, что ***multimap*** не поддерживает индексацию по ключу так, как это делает ***map***. Операции ***equal_range***(), ***lower_bound***() и ***upper_bound***() являются (§17.4.1.6) основными средствами доступа ко множественным значениям по единственному заданному ключу.

Естественно, для случаев, когда несколько значений могут соответствовать единственному ключу, ***multimap*** более предпочтителен, чем ***map***. В некоторых отношениях, ***multimap*** даже чище и элегантнее, чем ***map***.

Поскольку у человека запросто может быть несколько телефонных номеров, хорошим примером применения ***multimap*** является телефонная книга. Распечатать номера телефонов из телефонной книги можно, например, следующим образом:

```
void print_numbers(const multimap<string,int>& phone_book)
{
  typedef multimap<string,int>::const_iterator I;

  pair<I,I> b = phone_book.equal_range("Stroustrup");
  for(I i = b.first; i != b.second; ++i) cout<< i->second << '\n';
}
```

Для ***multimap*** аргумент операции ***insert***() всегда успешно вставляется в контейнер. Поэтому ***multimap***::***insert***() возвращает итератор, а не пару ***pair<iterator, bool>***, как в случае с ***map***. Ради однородности библиотека могла бы обеспечить для контейнеров ***map*** и ***multimap*** общую форму операции ***insert***(), несмотря на то, что ***bool*** был бы для ***multimap*** избыточным. Еще одно проектное решение могло бы предоставить простую операцию ***insert***(), которая не возвращала бы ***bool*** для обоих контейнеров, а пользователям ***map*** предоставляла бы какой-нибудь иной способ сигнализации об успешности операции. Это тот случай, когда разные проектные идеи вступают друг с другом в противоречие.

17.4.3. Ассоциативный контейнер set

Ассоциативные контейнеры ***set*** (*множества*) можно рассматривать как ассоциативные массивы (§17.4.1), для которых значения не важны и отслеживаются лишь ключи. Это приводит к незначительным изменениям в пользовательском интерфейсе:

```
template<class Key, class Cmp = less<Key>, class A = allocator<Key> >
class std::set
{
public:
  // как map, за исключением:
  typedef Key value_type;    // сам ключ является значением
  typedef Cmp value_compare;
  // нет операции индексирования []
};
```

Определяя ***value_type*** как тип ключа (***key_type***; см. §17.4.1.1), мы совершаем небольшой трюк, позволяющий практически идентичному коду работать и с контейнерами ***map***, и с контейнерами ***set***.

Обратите внимание на то, что множества полагаются на операцию сравнения (по умолчанию <), а не на операцию равенства ==. Это означает, что эквивалентность элементов определяется через неравенство (§17.1.4.1), и что итерирование по контейнеру ***set*** выполняется строго в определенном порядке.

Как и ***map***, контейнер ***set*** предоставляет операции ==, !=, <,>,<=,>= и ***swap***().

17.4.4. Ассоциативный контейнер multiset

Контейнер ***multiset*** — это то же множество ***set***, но допускающее дублирование ключей:

```
template<class Key, class Cmp = less<Key>, class A = allocator<Key> >
class std::multiset
{
public:
  // как set, за исключением:
  iterator insert(const value_type&);    // возвращает iterator, а не pair
};
```

Операции ***equal_range***(), ***lower_bound***() и ***upper_bound***() являются (§17.4.1.6) основными средствами доступа ко множественным вхождениям ключа.

17.5. «Почти контейнеры»

Встроенные массивы (§5.2), строки типа ***string*** (глава 20), массивы ***valarray*** (§22.4) и битовые поля ***bitset*** (§17.5.3) содержат элементы и, следовательно, могут считаться контейнерами. Однако все они не дотягивают до стандартного интерфейса контейнеров, так что эти «почти контейнеры» не полностью взаимозаменяемы со стандартными контейнерами вроде ***vector*** или ***list***.

17.5.1. Строки типа string

Тип ***basic_string*** обеспечивает индексацию, итераторы произвольного доступа и большинство удобств, предоставляемых стандартными контейнерами (глава 20). Однако ***basic_string*** не допускает широкого выбора типа элементов. Он оптимизирован для использования в качестве строки символов и в типичном случае используется не так, как стандартные контейнеры.

17.5.2. Массивы valarray

Массив ***valarray*** (§22.4) — это вектор, оптимизированный для вычислений. Он не претендует на то, чтобы быть универсальным контейнером. Тип ***valarray*** предоставляет множество полезных численных операций, однако из стандартных контейнерных операций (§17.1.1) он предлагает только ***size***`()` и операцию индексации (§22.4.2). Указатель на элемент массива ***valarray*** является итератором произвольного доступа (§19.2.1).

17.5.3. Битовые поля bitset

Часто некоторые аспекты системы, например, состояние входного потока (§21.3.3), представляются в виде набора флагов, отражающих такие бинарные атрибуты, как «хорошо/плохо», «истина/ложь» или «включено/выключено». Язык С++ эффективно поддерживает концепцию небольших наборов признаков (флагов) через битовые операции над целыми числами (§6.2.4). К этим операциям относятся `&` (побитовое логическое «И»), `|` (побитовое логическое «ИЛИ»), `^` (побитовое «исключающее ИЛИ»), << (сдвиг влево) и >> (сдвиг вправо). Класс ***bitset***<***N***> обобщает эту концепцию и обеспечивает удобную работу с набором из ***N*** бит, индексируемых от ***0*** до ***N-1***, где ***N*** предоставляется во время компиляции. Для битовых наборов, не помещающихся в ***unsigned long int***, использовать ***bitset*** намного удобнее, чем непосредственно целые числа. Для небольших наборов битов с точки зрения эффективности возможны варианты. Если же вы хотите именовать биты, а не нумеровать их, то альтернативой будет применение контейнера ***map*** (§17.4.1), контейнера ***set*** (§17.4.3), перечислений (§4.8) или полей битов в структурах (§С.8.1).

Класс ***bitset***<***N***> — это массив из ***N*** бит. От отличается от ***vector***<***bool***> (§16.3.11) тем, что имеет фиксированный размер; от ***set*** (§17.4.3) — тем, что его биты индексируются целыми числами, а не ассоциируются со значениями; от обоих указанных контейнеров тем, что предоставляет операции для манипуляций с битами.

Так как невозможно адресовать отдельный бит непосредственно с помощью встроенного указателя (§5.1), ***bitset*** вводит тип, ссылающийся на биты. Это универсальная технология адресации объектов, для которых применение встроенных указателей по той или иной причине не подходит:

```
template<size_t N> class std::bitset
{
public:
  class reference                              // ссылка на одиночный бит:
  {
   friend class bitset;
   reference();

  public:                                      // b[i] ссылается на (i+1)-й бит:
   ~reference();
   reference& operator=(bool x);               // для b[i] = x;
   reference& operator=(const reference&);     // для b[i] = b[j];
   bool operator~() const;                     // возвращает ~b[i]
   operator bool() const;                      // для x = b[i];
   reference& flip();                          // b[i].flip();
  };
  //...
};
```

Шаблон ***bitset*** определен в пространстве имен ***std*** и расположен в заголовочном файле <***bitset***>.

По историческим причинам ***bitset*** отличается от других классов стандартной библиотеки по стилю. Например, если индекс (известный как *битовая позиция — bit position*) выходит за штатные границы, генерируется исключение ***out_of_range***. Никаких итераторов для него нет. Битовые позиции отсчитываются справа налево (как нумеруются разряды чисел), так что величина ***b[i]*** равна ***pow(2, i)***. Таким образом, битовые поля можно трактовать как двоичные числа из ***N*** бит:

позиция:	9	8	7	6	5	4	3	2	1	0
bites<10>(989):	1	1	1	1	0	1	1	1	0	1

17.5.3.1. Конструкторы

Битовые поля могут конструироваться с умолчательными значениями, из битов чисел типа ***unsigned long int***, или из строк типа ***string***:

```
template<size_t N> class bitset
{
public:
  //...
  // конструкторы:
  bitset();                                        // N нулевых бит
  bitset(unsigned long val);                       // биты из val

  template<class Ch, class Tr, class A>            // Tr - character trait (§20.2)
  explicit bitset(const basic_string<Ch,Tr,A>& str,  // биты из str
           typename basic_string<Ch,Tr,A>::size_type pos = 0,
           typename basic_string<Ch,Tr,A>::size_type
 n=basic_string<Ch,Tr,A>::npos);
  //...
};
```

Умолчательное значение бита равно ***0***. Когда задается аргумент типа ***unsigned long int***, каждый бит этого числа используется для инициализации соответствующего бита в битовом поле (если таковой есть). То же самое делает и ***basic_string*** (глава 20), у которого символ ***'0'*** дает битовое значение ***0***, символ ***'1'*** дает битовое значение ***1***, а все остальные символы приводят к генерации исключения ***invalid_argument***. По умолчанию вся строка используется для инициализации. Однако в конструкторах ***basic_string*** (§20.3.4) можно указать диапазон используемых символов (от ***pos*** до конца символов или от ***pos*** до ***pos+n***). Например:

```
void f()
{
  bitset<10> b1;                                  // все 0

  bitset<16> b2 = 0xaaaa;                         // 1010101010101010
  bitset<32> b3 = 0xaaaa;                         // 00000000000000001010101010101010

  bitset<10> b4(string("1010101010"));            // 1010101010
  bitset<10> b5(string("10110111011110"),4);      // 0111011110
  bitset<10> b6(string("10110111011110"),2,8);    // 0011011101

  bitset<10> b7(string("n0g00d"));   // invalid_argument (недопустимый аргумент)
  bitset<10> b8 = string("n0g00d");  // error: нет приведения из string в bitset conversion
}
```

Ключевая идея, с которой спроектирован ***bitset***, заключается в возможности предоставления оптимизированной реализации для случаев, когда набор битов умещается в одно машинное слово. Интерфейс отражает такое допущение.

17.5.3.2. Побитовые операции

Шаблон ***bitset*** предоставляет битовые операции для доступа к отдельным битам и для манипуляции всеми битами в наборе:

```
template<size_t N> class std::bitset
{
public:
  // ...
  // битовые операции:
  reference operator[](size_t pos);            // b[i]

  bitset& operator&=(const bitset& s);         // and (И)
  bitset& operator|=(const bitset& s);         // or (ИЛИ)
  bitset& operator^=(const bitset& s);         // exclusive or (ИСКЛЮЧАЮЩЕЕ ИЛИ)

  bitset& operator<<=(size_t n);               // сдвиг влево (с заполнением нулями)
  bitset& operator>>=(size_t n);               // сдвиг вправо (с заполнением нулями)

  bitset& set();                               // установка всех битов в 1
  bitset& set(size_t pos, int val= 1);         // b[pos]=val

  bitset& reset();                             // установка всех битов в 0
  bitset& reset(size_t pos);                   // b[pos]=0

  bitset& flip();                              // изменение значения каждого бита
  bitset& flip(size_t pos);                    // изменяет значение b[pos]

  bitset operator~() const {return bitset<N>(*this).flip();}  // дополнительные биты
  bitset operator<<(size_t n) const {return bitset<N>(*this)<<=n;} // сдвинутые биты
```

```
    bitset operator>> (size_t n) const {return bitset<N> (*this) >>=n; } // сдвинутые биты
    // ...
};
```

Операция индексирования генерирует исключение *out_of_range*, когда индекс выходит за границы набора битов. Индексации без проверки индекса нет.

У этих операций возврат *bitset&* — это **this*. Операции, возвращающие *bitset* (а не *bitset&*), делают копию **this*, выполняют операцию над копией и возвращают результат. Здесь << и >> есть операции битового сдвига, а не операции ввода/вывода. Операция вывода для *bitset* есть <<, принимающая аргументы типа *ostream* и *bitset* (§17.5.3.3).

При *сдвиге битов* применяется *логический* (а *не циклический*) *сдвиг*. Это означает, что некоторые биты пропадают на конце, а часть битов заполняется умолчательными нулями. Поскольку *size_t* есть беззнаковый тип, то невозможно выполнить сдвиг на отрицательное число, и выражение *b<<-1* на самом деле означает сдвиг на очень большое положительное число. Это фактически обнуляет все биты *b*. Желательно, чтобы компилятор выдавал в таких случаях предупреждение.

17.5.3.3. Прочие операции

Шаблон *bitset* поддерживает такие стандартные операции, как *size()*, ==, *I/O* и т.п.:

```
template<size_t N> class bitset
{
public:
    // ...
    unsigned long to_ulong() const;

    template <class Ch, class Tr, class A> basic_string<Ch,Tr,A> to_string() const;

    size_t count() const;                  // число бит со значением 1
    size_t size() const {return N; }       // число бит

    bool operator== (const bitset& s) const;
    bool operator!= (const bitset& s) const;

    bool test(size_t pos) const;           // true если b[pos] равно 1
    bool any() const;                      // true если любой бит равен 1
    bool none() const;                     // true если никакой бит не равен 1
};
```

Функции *to_ulong()* и *to_string()* предоставляют операции, обратные конструированию. Такие именованные функции были выбраны вместо стандартных операций преобразования, чтобы избежать возникновения неочевидных преобразований. Если значение bitset имеет так много значащих битов, что его невозможно представить в виде *unsigned long*, то операция *to_ulong()* генерирует исключение *overflow_error*.

Операция *to_string()* производит строку выбранного типа, содержащую символы *'0'* и *'1'*; *basic_string* — это шаблон, используемый для реализации строк (глава 20). Операцию *to_string()* можно использовать, например, для генерации бинарного представления целых чисел:

```
void binary(int i)
{
  bitset<8*sizeof(int)> b = i;          // полагаем 8-bit byte (см. также §22.2)

  cout<< b.to_string<char,char_traits<char>,allocator<char> >() << '\n';
}
```

К сожалению, вызов явно квалифицируемых шаблонных функций-членов требует редкого синтаксиса (§C.13.6).

Еще *bitset* перегружает бинарные операции &, |, ^, а также обычные операции ввода/вывода:

```
template<size_t N> bitset<N> std::operator&(const bitset<N>&, const bitset<N>&);
template<size_t N> bitset<N> std:operator|(const bitset<N>&, const bitset<N>&);
template<size_t N> bitset<N> std::operator^(const bitset<N>&, const bitset<N>&);

template<class charT, class Tr, size_t N>
basic_istream<charT,Tr>& std::operator>>(basic_istream<charT,Tr>&, bitset<N>&);
template<class charT, class Tr, size_t N>
basic_ostream<charT,Tr>& std::operator<<(basic_ostream<charT,Tr>&, const
bitset<N>&);
```

Поэтому мы можем вывести битовый набор без преобразования его в строку. Например:

```
void binary(int i)
{
  bitset<8*sizeof(int)> b = i;   // полагаем 8-bit byte (см. также §22.2)
  cout<< b << '\n';
}
```

Этот код выводит биты, представляя их символами *'0'* и *'1'* слева направо (старший бит — самый левый).

17.5.4. Встроенные массивы

Встроенные массивы обеспечивают индексацию и итераторы с произвольным доступом в виде обычных указателей (§2.7.2). Однако встроенные массивы не знают собственных размеров, поэтому следить за их размерами должны пользователи. Кроме того, у встроенных массивов, в отличие от стандартных контейнеров, нет ни функций-членов, ни определяемых ими типов.

Возможно, а иногда и полезно, снабдить обычный массив удобствами стандартного контейнера без изменения его низкоуровневой сущности:

```
template<class T, int max> struct c_array
{
  typedef T value_type;

  typedef T* iterator;
  typedef const T* const_iterator;
  typedef T& reference;
  typedef const T& const_reference;

  T v[max];
  operator T*() {return v;}

  reference operator[](ptrdiff_t i) {return v[i];}
  const_reference operator[](ptrdiff_t i) const {return v[i];}
```

```
    iterator begin() { return v; }
    const_iterator begin() const { return v; }
    iterator end() { return v+max; }
    const_iterator end() const { return v+max; }

    size_t size() const { return max; }
};
```

Для совместимости с обычными массивами я воспользовался знаковым типом ***ptrdiff_t*** (§16.1.2), а не беззнаковым ***size_t*** в качестве типа индекса. Применение ***size_t*** могло привести к тонким неоднозначностям при использовании операции [] с объектами типа ***c_array***.

Шаблон ***c_array*** не входит в стандартную библиотеку. Он приведен в качестве примера того, как можно представить чужеродный контейнер под личиной стандартного контейнера. Его можно использовать со стандартными алгоритмами (глава 18), применяя ***begin***(), ***end***() и т.д. Он может быть размещен непосредственно в стеке без какого-либо косвенного выделения динамической памяти. Его можно передать написанной в С-стиле функции, ожидающей обычного указателя. Например:

```
void f(int* p, int sz);    // функция в С-стиле

void g()
{
    c_array<int,10> a;
    f(a,a.size());              // используем функцию в С-стиле

    c_array<int,10>::iterator p = find(a.begin(),a.end(),777);    // C++/STL стиль
    //...
}
```

17.6. Создание нового контейнера

Стандартные контейнеры составляют полезный каркас (среду), к которому пользователь может добавлять свои собственные средства. Здесь я покажу создание контейнера, взаимозаменяемого со стандартными контейнерами. Его устройство будет вполне реалистичным, хотя и не оптимальным. Интерфейс будет близок к существующим, широко применяемым и высококачественным реализациям понятия ***hash_map***. Используйте его для изучения, а на практике применяйте стандартные реализации.

17.6.1. Контейнер hash_map

Контейнер ***map*** — это ассоциативный контейнер, допускающий элементы почти что любого типа. Он обеспечивает эту возможность, опираясь лишь на операцию < для сравнения элементов (§17.4.1.5). Однако если мы знаем больше о типе ключа, то мы можем ускорить поиск элементов, предоставив хэш-функцию и реализовав контейнер как хэш-таблицу.

Хэш-функция (*hash-function*) — это функция, которая быстро преобразует значение в индекс таким образом, что *два различных значения редко когда приводят к одинаковому индексу*. Хэш-таблицы стандартно формируются размещением значения по его индексу, или «рядом» (если там уже размещено какое-то значение). Поиск

элемента, расположенного по своему индексу, занимает мало времени, а поиск элемента, расположенного «рядом», не намного больше, если конечно эффективно выполняется проверка на равенство. Как следствие, не редки случаи, когда ***hash_map*** обеспечивает в 5—10 раз более быстрый поиск для больших контейнеров, чем это делает ***map***. В то же время, если хэш-функция подобрана плохо, то ***hash_map*** может оказаться более медленным, чем ***map***.

Существует много способов реализовать хэш-таблицу. Интерфейс ***hash_map*** обычно строится таким образом, чтобы он отличался от интерфейса стандартных ассоциативных контейнеров только тогда, когда это нужно для выигрыша в быстродействии за счет хэширования. Самое важное отличие ***hash_map*** от ***map*** состоит в том, что ***map*** требует от типа своих элементов операцию <, а ***hash_map*** — операцию == и хэш-функцию. Таким образом, ***hash_map*** отличается от ***map*** созданием объектов «не по умолчанию». Например:

```
map<string, int> m1;               // сравнение строк при помощи <
map<string, int, Nocase> m2;       // сравнение строк при помощи Nocase()(§17.1.4.1)

hash_map<string, int> hm1;         // хэшируем с Hash<string>() (§17.6.2.3)
hash_map<string, int, hfct> hm2;   // хэшируем с hfct(); сравнение через операцию ==
hash_map<string, int, hfct, eql> hm3; // хэшируем с hfct(), сравнение через eql
```

Контейнер, использующий хэшированный поиск, реализуется при помощи одной или нескольких таблиц. Кроме хранения элементов контейнер должен следить за тем, какие значения ассоциированы с хэш-значениями («индексами», согласно приведенному выше пояснению); это делается при помощи «хэш-таблицы». Для большинства реализаций хэш-таблиц производительность резко падает при увеличении плотности заполнения таблицы (скажем, на 75% и более). Из-за этого ***hash_map*** автоматически изменяет свой размер при увеличении плотности заполнения. Так как изменение размера является затратной операцией, целесообразно задавать подходящий начальный размер. В результате в первом приближении ***hash_map*** выглядит следующим образом:

```
template<class Key, class T, class H = Hash<Key>,
        class EQ = equal_to<Key>, class A = allocator<pair<const Key, T> > >
class hash_map
{
  // как map, за исключением:
  typedef H Hasher;
  typedef EQ key_equal;

  hash_map(const T& dv=T(), size_type n =101, const H& hf=H(), const EQ& =EQ());

  template<class In> hash_map(In first, In last, const T& dv =T(), size_type n =101,
                              const H& hf=H(), const EQ& =EQ());
};
```

В своей основе это интерфейс контейнера ***map*** (§17.4.1.4), за исключением того, что < заменена на == и добавлена хэш-функция.

Имеющиеся в данной книге примеры использования ***map*** (§3.7.4, §6.1, §17.4.1) могут быть переконвертированы на применение ***hash_map*** простейшей заменой имени ***map*** на ***hash_map***. Это можно упростить еще сильнее, используя оператор ***typedef***. Например:

```
typedef hash_map<string, record> Map;
Map dictionary;
```

Оператор *typedef* позволяет скрыть от пользователей точный тип словаря.

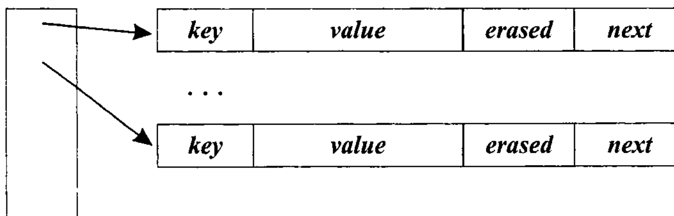
Хотя это и не совсем точно, но я представляю себе альтернативу *map/hash_map* как альтернативу память/время. Если эффективность не является главной проблемой, не тратьте время на выбор — хорошо подойдет любой из этих контейнеров. Для больших и интенсивно применяемых таблиц *hash_map* имеет явное преимущество в скорости и должен использоваться во всех случаях, когда нет недостатка в памяти. И даже в последнем случае я не стал бы так сразу переключаться на *map*, не рассмотрев дополнительных возможностей экономии памяти (требуются точные замеры для выбора оптимального варианта).

Эффективное хэширование достигается лишь с помощью качественных хэш-функций. При не слишком качественной хэш-функции *map* может легко превзойти *hash_map* по быстродействию. Обычно, хэширование на базе С-строк, строк типа *string* или целых чисел весьма эффективно. Однако нельзя забывать, что эффективность хэш-функций критически зависит от самих хэшируемых значений (§17.8[35]). Контейнер *hash_map* должен использоваться в случаях, когда операция < не определена или неприменима к выбранному типу ключа. Противоположная рекомендация — контейнер *map* нужно использовать тогда, когда требуется держать элементы строго упорядоченным образом, ибо хэш-функция не задает упорядочения так, как это делает операция <.

Так же как и *map*, *hash_map* предоставляет функцию *find*(), позволяющую программисту выяснить, содержится ли данный ключ в контейнере.

17.6.2. Представление и конструирование

Для *hash_map* возможны разные реализации. Здесь я привожу такую реализацию, которая обеспечивает неплохую скорость и простоту наиболее важных опера-



ций. Важнейшие из них — конструкторы, поиск (операция []), изменение размеров и удаление элемента (*erase*()).

Выбранная нами простая реализация основана на хэш-таблице, представляющей собой вектор указателей на записи. Каждая запись содержит ключ (*key*), значение (*value*), указатель на следующую (если имеется) запись с тем же хэш-значением, и флаг *erased* (удален):

В объявлении это выглядит следующим образом:

```
template<class Key, class T, class H = Hash<Key>,
        class EQ = equal_to<Key>, class A = allocator<pair<const Key, T> > >
```

```
class hash_map
{
  // ...
private:                        // внутреннее представление
  struct Entry
  {
    key_type key;
    mapped_type val;
    bool erased;

    Entry* next;

    Entry (key_type k, mapped_type v, Entry* n)
     : key (k), val (v), erased (false), next (n) { }
  };

  vector<Entry> v;              // истинные входы
  vector<Entry*> b;             // хэш-таблица: указатели внутрь v
  // ...
  };
};
```

Отметим поле ***erased***. То, как здесь обрабатываются несколько значений с единственным хэш-значением, затрудняет удаление элемента. Поэтому вместо действительного удаления при вызове ***erase***() я просто помечаю элемент как ***erased*** (удален) и игнорирую его пока таблица не изменит размер.

В дополнение к главной структуре данных контейнер ***hash_map*** нуждается еще и в некотором количестве административных данных. Естественно, каждый конструктор должен проинициализировать все эти данные. Например:

```
template<class Key, class T, class H = Hash<Key>,
        class EQ = equal_to<Key>, class A = allocator<pair<const Key, T> > >
class hash_map
{
  // ...
  hash_map (const T& dv =T (), size_type n =101, const H& h =H (), const EQ& e =EQ () )
     : default_value (dv), b{n), no_of_erased (0), hash (h}, eq (e)
  {
    set_load ();                          // все, что по умолчанию
    v.reserve (max_load*b.size () );      // резервирует память для роста
  }

  void set_load (float m=0.7, float g=1.6) {max_load = m; grow = g; }
  // ...

private:
  float max_load;               // сохраняем v.size()<=b.size ()*max_load
  float grow;                   // при необходимости меняем размер, resize(bucket_count()*grow)
  size_type no_of_erased;       // количество входов в v, занятых стертыми элементами
  Hasher hash;                  // хэш-функция
  key_equal eq;                 // равенство

  const T default_value;        // умолчательное значение, используемое операцией []
};
```

Стандартные ассоциативные контейнеры требуют, чтобы отображенный тип имел умолчательное значение (§17.4.1.7). Такое ограничение не является логически необходимым и может причинять неудобства. Указывая умолчательное значение в качестве аргумента, можно написать:

```
hash_map<string,Number> phone_book1;
hash_map<string,Number> phone_book2(Number(411));
```

17.6.2.1. Поиск

Наконец мы можем ввести важнейшие операции поиска:

```
template<class Key, class T, class H = Hash<Key>,
        class EQ = equal_to<Key>, class A = allocator<pair<const Key, T> > >
class hash_map
{
  // ...
  mapped_type& operator[](const key_type& k);

  iterator find(const key_type&);
  const_iterator find(const key_type&) const;
  // ...
};
```

Чтобы найти *value*, операция `[]` использует хэш-функцию с целью нахождения индекса в хэш-таблице для ключа (*key*). Затем просматриваются записи до тех пор, пока не будет найден совпадающий *key*. Значение *value* в этой записи и есть то, что мы ищем. Если оно не находится, вставляется умолчательное значение:

```
template<class Key, class T, class H = Hash<Key>,
        class EQ = equal_to<Key>, class A = allocator<pair<const Key , T> > >
typename hash_map<Key,T,H,EQ,A>::mapped_type&
hash_map<Key,T,H,EQ,A>::operator[](const key_type& k)
{
  size_type i = hash(k)%b.size();                  // хэш

  for(Entry* p = b[i]; p; p = p->next)             // поиск среди входов, хэшированных в i
    if(eq(k,p->key))                               // найдено
    {
      if(p->erased)                                // повторная вставка
      {
        p->erased = false;
        no_of_erased--;
        return p->val = default_value;
      }
      return p->val;
    }

  // не найдено:
  if(size_type(b.size()*max_load <=v.size()))      // слишком плотное заполнение
  {
    resize(b.size()*grow);                         // расширяем
    return operator[](k);                          // рехэшируем
  }
```

```
    v.push_back(Entry(k,default_value,b[i])); // добавляем Entry
    b[i] = &v.back();                          // указываем на новый элемент

    return b[i]->val;
}
```

В отличие от ***map***, ***hash_map*** не опирается на проверку эквивалентности, синтезированную из операции сравнения на меньше (§17.1.4.1). Это связано с вызовом функции ***eq***() в цикле просмотра элементов с одинаковым хэш-значением. Данный цикл критически важен для производительности поиска, а для обычных типов ключа, таких как ***string*** или С-строка, перерасход времени на лишние сравнения может быть значительным.

Я мог бы использовать ***set<Entry>*** для представления множества записей с одинаковыми хэш-значениями. Однако если у нас есть хорошая хэш-функция (***hash***()) и хэш-таблица подходящего размера (***b***), то большинство таких множеств будут содержать ровно один элемент. Поэтому я связал элементы этого множества между собой с помощью поля ***next*** каждой записи ***Entry*** (§17.8[27]).

Заметим, что ***b*** содержит указатели на элементы вектора ***v*** и что элементы добавляются в ***v***. Вообще говоря, функция ***push_back***() может вызвать перераспределение памяти и, тем самым, сделать указатели недействительными (неверными) (§16.3.5). Однако в данном случае конструкторы (§17.6.2) и функция ***resize***() применяют ***reserve***() с целью аккуратного резервирования памяти, предотвращающего ее неожиданные перераспределения.

17.6.2.2. Операции erase() и resize()

Хэшированный поиск становится неэффективным, когда таблица переполняется. Чтобы снизить вероятность этого, таблица автоматически изменяет размер при помощи функции ***resize***(), вызываемой из операции индексации. Операция ***set_load***() (§17.6.2) обеспечивает контроль за этим процессом (как и когда происходит изменение размера таблицы). Другие функции позволяют программисту следить за состоянием ***hash_map***:

```
template<class Key, class T, class H = Hash<Key>,
        class EQ = equal_to<Key>, class A = allocator<pair<const Key ,T> > >
class hash_map
{
  // ...
  void resize(size_type n);                                  // размер хэш-таблицы - в n

  void erase(iterator position);                             // удаление указуемого эл-та

  size_type size() const {return v.size()-no_of_erased;}     // число элементов

  size_type bucket_count() const {return b.size();}          // размер хэш-таблицы

  Hasher hash_fun() const {return hash;}                     // применяемая хэш-функция
  key_equal key_eq() const {return eq;}                      // равенство
  // ...
};
```

Операция ***resize***() очень важна, достаточно проста и потенциально дорога:

```
template<class Key, class T, class H = Hash<Key>,
        class EQ = equal_to<Key>, class A = allocator<pair<const Key,T> > >
void hash_map<Key,T,H,EQ,A>::resize(size_type s)
{
  size_type i = v.size();

  if(s <= b.size()) return;

  while(no_of_erased)        // реально устраняет "удаленные" элементы
  {
    if(v[--i].erased)
    {
      v.erase(v.begin()+i);
      --no_of_erased;
    }
  }

  b.resize(s);
  fill(b.begin(),b.end(),0);      // обнуляем входы (§18.6.6)
  v.reserve(s*max_load);          // если v нуждается в памяти, делаем это сейчас

  for(size_type i = 0; i<v.size(); i++)       // рехэширование
  {
    size_type ii = hash(v[i].key)%b.size();   // хэширование
    v[i].next = b[ii];                        // связь
    b[ii] = &v[i];
  }
}
```

При необходимости пользователь может «вручную» вызвать ***resize***(), чтобы избавиться от неожиданных вызовов этой функции. Функция ***resize***() крайне важна во многих приложениях, но она не является принципиально фундаментальной для хэш-таблиц. В некоторых реализациях она вообще не применяется.

Вся реальная работа делается в другом месте (и только если ***hash_map*** изменяет размеры), так что функция ***erase***() тривиальна:

```
template<class Key, class T, class H = Hash<Key>,
        class EQ = equal_to<Key>, class A = allocator<pair<const Key,T> > >
void hash_map<Key,T,H,EQ,A>::erase(iterator p)
{
  if(p->erased == false) no_of_erased++;
  p->erased = true;
}
```

17.6.2.3. Хэширование

Чтобы закончить с ***hash_map***::***operator***[](), нам еще нужно определить функции ***hash***() и ***eq***(). По причинам, которые станут яснее в §18.4, хэш-функцию лучше всего определить в виде функции ***operator***()() класса функциональных объектов:

```
template<class T> struct Hash : unary_function<T,size_t>
{
  size_t operator()(const T& key) const;
};
```

Хорошая хэш-функция берет ключ и возвращает целое число таким образом, что различные ключи с высокой степенью вероятности отвечают различным числам. Выбор хорошей хэш-функции — это искусство. Часто, однако, вполне приемлемой является операция «исключающее ИЛИ» над битовым представлением ключа:

```
template<class T> size_t Hash<T>::operator() (const T& key) const
{
  size_t res = 0;

  size_t len = sizeof(T);
  const char* p = reinterpret_cast<const char*>(&key);

  while (len--) res = (res<<1)^*p++;
  return res;
}
```

Применение ***reinterpret_cast*** (§6.2.7) служит четким указанием на то, что выполняется нечто исключительное, и что в случаях, когда о хэшируемых объектах имеется более подробная информация, можно поступить лучше. В частности, если объекты содержат указатели, являются очень большими или имеют внутри пустые места из-за требований к выравниванию их полей, мы можем что-нибудь улучшить (см. §17.8[29]).

С-строка — это указатель, а строки ***string*** содержат указатели. Следовательно, уместны специализации:

```
typedef char* Pchar;

template<> size_t Hash<Pchar>::operator() (const Pchar& key) const
{
  size_t res = 0;

  Pchar p = key;
  while (*p) res = (res<<1)^*p++;
  return res;
}

template<> size_t Hash<string>::operator() (const string& key) const
{
  size_t res = 0;
  typedef string::const_iterator CI;

  CI p = key.begin();
  CI end = key.end();

  while (p!=end) res = (res<<1)^*p++;
  return res;
}
```

Реализация ***hash_map*** должна включать хэш-функции по крайней мере для целых и строковых ключей. Для других типов ключей пользователю придется прибегнуть к иным специализациям. При выборе хэш-функций большое значение имеют эксперимент и замер результатов. Интуиция в этой области работает плохо.

Для завершения контейнера ***hash_map*** требуется определить еще итераторы и большое число вспомогательных функций; оставляем это в качестве упражнения (§17.8[34]).

17.6.3. Другие хэшированные ассоциативные контейнеры

Для полноты и согласованности помимо ***hash_map*** должны быть еще ***hash_set***, ***hash_multimap*** и ***hash_multiset***. Их определения вытекают очевидным образом из определений ***hash_map***, ***map***, ***multimap***, ***set*** и ***multiset***, так что я оставляю это в качестве упражнения (§17.8[34]). Доступны хорошие коммерческие и свободно распространяемые реализации этих хэшированных ассоциативных контейнеров. Для реальных применений их следует предпочесть версиям вроде моих, сконцентрированных на локальных проблемах.

17.7. Советы

1. Если вам нужен контейнер, по умолчанию используйте ***vector***; §17.1.
2. Узнайте цену (сложность, асимптотическую О-оценку) каждой часто используемой вами операции; §17.1.2.
3. Интерфейс, реализация и представление контейнеров — различные понятия. Не путайте их; §17.1.3.
4. Выполнять сортировку и поиск можно в соответствии с разными критериями; §17.1.4.1.
5. Не пользуйтесь С-строками в качестве ключа, если вы не предоставили соответствующего критерия сравнения; §17.1.4.1.
6. Вы можете определить критерий сравнения так, чтобы эквивалентные, хотя и различающиеся значения ключей, отображались в один и тот же ключ; §17.1.4.1.
7. При вставке или удалении элементов предпочитайте операции на конце контейнера (***back***-операции);§17.1.4.1.
8. Если нужно выполнять много вставок и удалений в голове или середине контейнера, предпочитайте ***list***; §17.2.2.
9. Используйте ***map*** или ***multimap***, когда обращение к элементам выполняется главным образом по ключу; §17.4.1.
10. Для достижения максимальной гибкости используйте минимальный набор операций; §17.1.1.
11. Если элементы должны быть упорядочены, выбирайте ***map***, а не ***hash_map***; §17.6.1.
12. Когда важна скорость поиска, выбирайте ***hash_map***, а не ***map***; §17.6.1.
13. Если для элементов невозможно определить операцию <, выбирайте ***hash_map***, а не ***map***; §17.6.1.
14. Для проверки наличия ключа в ассоциативном контейнере применяйте функцию ***find***() ; §17.4.1.6.

15. Для нахождения всех элементов контейнера с заданным ключом применяйте функцию ***equal_range***(); §17.4.1.6.
16. Если нужно хранить несколько значений для одного ключа, применяйте ***multimap***; §17.4.2.
17. Когда сам ключ является единственным значением, которое нужно хранить, применяйте ***set*** или ***multiset***; §17.4.3.

17.8. Упражнения

Решение некоторых задач можно найти, посмотрев реализацию стандартной библиотеки. Сделайте себе полезное одолжение: не пытайтесь смотреть сторонний код до того, как вы сами попробовали решить задачу.

1. (*2.5) Изучите *O*-нотацию (§17.1.2). Выполните измерения для операций стандартных контейнеров с целью определения числовых коэффициентов, вовлеченных в *O*-нотацию.
2. (*2) Существуют телефонные номера, которые не умещаются в тип ***long***. Напишите тип ***phone_number*** и класс, предоставляющий набор полезных операций с контейнером телефонных номеров типа ***phone_number***.
3. (*2) Напишите программу, которая заносит разные слова в файл в алфавитном порядке. Создайте две версии: в первой из них слова есть последовательность символов, ограниченных пробельными символами, а во второй слова есть последовательность букв, ограниченных символами, не являющихся буквами.
4. (*2.5) Реализуйте простую версию карточной игры «Solitaire».
5. (*1.5) Реализуйте простой тест слов, выявляющий, являются ли они палиндромами (например, *казак*, *ada*, *odo* и т.д.). Также реализуйте простой тест целых чисел, выявляющий, являются ли эти числа палиндромами. Затем реализуйте проверку на палиндромность уже целых предложений. Обобщите эти решения.
6. (*1.5) Определите очередь (queue), используя (только) два ***стека***.
7. (*1.5) Определите стек, похожий на тип ***stack*** (§17.3.1), но который не копирует нижележащий контейнер (на котором он базируется) и который допускает итерации по своим элементам.
8. (*3) Понятия потока, задачи и процесса составляют основные понятия параллельного исполнения программ на вашем компьютере. Разберитесь подробнее в этих механизмах. Для предотвращения одновременного доступа двух задач к одной области памяти применяется блокировка. Реализуйте класс блокировки, опираясь на системный механизм блокировок на вашей машине.
9. (*2.5) Читайте даты из потока ввода, например, ***Dec85***, ***Dec50***, ***Jan76*** и т.д., а потом выведите их так, чтобы более поздние даты шли первыми. Формат дат должен состоять из трех символов под месяц, после чего следуют два символа под год. Полагаем, что все годы относятся к одному веку.

10. (*2.5) Обобщите входной формат для дат так, чтобы он включал даты типа ***Dec1985***, ***12/3/1990***, ***3/6/2001*** и т.д. Переделайте упражнение §17.8[9], чтобы оно соответствовало новому формату.
11. (*1.5) Используйте ***bitset*** для печати двоичных значений некоторых чисел, например ***1***, ***-1***, ***0***, ***18*** и ***-18***, а также максимально возможного положительного ***int***.
12. (*1.5) Используйте ***bitset*** для хранения информации о том, кто в текущий день присутствует на занятиях. Прочтите эти ***bitset*** за 12 дней и определите, кто присутствовал всегда? Кто присутствовал не менее 8 дней?
13. (*1.5) Определите список указателей, который уничтожает объекты, адресуемые этими указателями, во время уничтожения самого списка или при удалении элемента из списка операцией ***remove***.
14. (*1.5) Располагая объектом типа ***stack***, распечатайте по порядку его элементы (не изменяя содержимого самого стека).
15. (*2.5) Завершите шаблон ***hash_map*** (§17.6.1), то есть реализуйте функции ***find***() и ***equal_range***(), а также продумайте схему его тестирования. Выявите хотя бы один тип ключа, для которого хэш-функция у типа ***hash_map*** плохо подходит (так что требуется иная хэш-функция).
16. (*2.5) Реализуйте и оттестируйте работу некоторого списка, выполненного в духе стандартного типа ***list***.
17. (*2) Иногда бывает, что излишнее потребление списком ***list*** памяти накладно. Напишите и оттестируйте работу односвязного списка, выполненного в духе стандартных контейнеров.
18. (*2.5) Реализуйте список, похожий на стандартный ***list***, но дополнительно поддерживающий индексацию. Сравните стоимость индексации списков и стоимость индексации для стандартного вектора (то есть типа ***vector***).
19. (*2) Реализуйте шаблонную функцию, которая выполняет слияние двух контейнеров.
20. (*1.5) Располагая С-строкой, определите, не является ли она палиндромом. Определите, не является ли палиндромом ее начальная последовательность из хотя бы трех слов?
21. (*2) Прочтите последовательность пар (имя,значение) и сформируйте отсортированный список четверок (*имя*, *сумма*, *среднее значение*, *медиана*). Распечатайте этот список.
22. (*2.5) Определите расход памяти под стандартные контейнеры на вашей системе.
23. (*3.5) Рассмотрите вопрос об оптимальной стратегии реализации ***hash_map***, требующей минимального расхода памяти под такой ассоциативный контейнер. Затем рассмотрите вопрос об оптимальной стратегии реализации ***hash_map***, достигающей минимального времени поиска. В обоих случаях решите, какими операциями пренебречь ради приближения к идеалу. Подсказка: имеется огромная литература, посвященная хэшированию.
24. (*2) Разработайте такую стратегию обработки переполнения в ***hash_map*** (слишком высокая кучность хэшей для разных значений), при которой реализация ***equal_range***() была бы тривиальной.

25. (*2.5) Оцените затраты памяти под ***hash_map***, а затем измерьте эти затраты. Сравните оценку с результатом измерений.

26. (*2.5) Постарайтесь выявить, где сосредоточены основные временные затраты в вашем ***hash_map***, и каковы они. Сравните эти результаты с аналогичными результатами для стандартного ***map***, а также для какого-либо иного ***hash_map***.

27. (*2.5) Реализуйте ***hash_map***, основанный на ***vector<map<K, V>*>***, так, чтобы каждый ***map*** содержал все ключи с одинаковым хэш-значением.

28. (*3) Реализуйте ***hash_map***, используя Splay-деревья (см. D. Sleator, R.E. Tarjan, *Self-Adjusting Binary Search Trees*, JASM, Vol. 32, 1985).

29. (*2) Дана структура данных, описывающая строкоподобную сущность:

```
struct St
{
  int size;
  char type_indicator;
  char* buf;              // указывает на size символов
  St(const char* p) ;    // выделяет память под буфер и заполняет его
}
```

Создайте 1000 объектов типа ***St*** и используйте их в качестве ключей для ***hash_map***. Измерьте производительность такого ***hash_map***. Напишите хэш-функцию (***Hash***; §17.6.2.3) специально для ключей ***St***.

30. (*2) Приведите четыре разных способа удаления из ***hash_map*** «стертых» (erased) элементов. Используйте стандартный библиотечный алгоритм (§3.8, глава 18) во избежание явного цикла.

31. (*3) Реализуйте ***hash_map*** с немедленным удалением элементов.

32. (*2) Хэш-функции, рассмотренные в §17.6.2.3, не всегда используют полное представление ключа. Когда часть представления игнорируется? Приведите пример, когда бывает разумным игнорировать часть ключа и напишите соответствующую хэш-функцию.

33. (*2.5) Код хэш-функций имеет обыкновение следовать общей схеме — в цикле получают новые данные и хэшируют их. Определите ***Hash*** (§17.6.2.3), который получает данные повторяющимся вызовом функции, которую пользователь предоставляет для каждого типа ключа. Например:

```
size_t res = 0;
while (size_t v=hash (key) ) res= (res<<3) ^v;
```

Здесь пользователь может определить ***hash(K)*** для каждого типа ***K***, подлежащего хэшированию.

34. (*3) Имеется некоторая реализация ***hash_map***. Реализуйте ***hash_multimap***, ***hash_set*** и ***hash_multiset***.

35. (*2.5) Напишите хэш-функцию, предназначенную для отображения равномерно распределенных целых значений в таблицу хэш-значений размером около 1024. Исходя из такой функции, придумайте набор из 1024 ключей, каждый из которых отображается в одно и то же значение.

18

Алгоритмы и классы функциональных объектов

Формальность освобождает.
— Популярная поговорка инженеров

Введение — обзор стандартных алгоритмов — последовательности — функциональные объекты — предикаты — арифметические функциональные объекты — связывающие адаптеры — адаптирование функций-членов — алгоритм ***for_each*** — поиск элементов — ***count*** — сравнение последовательностей — алгоритмы поиска — копирование — ***transform*** — замещение и удаление элементов — заполнение последовательности — изменение порядка — ***swap*** — сортированные последовательности — ***binary_search*** — ***merge*** — операции над множествами — ***min*** и ***max*** — кучи — перестановки — алгоритмы в С-стиле — советы — упражнения.

18.1. Введение

Контейнеры сами по себе не столь интересны. По настоящему полезными они становятся тогда, когда снабжаются операциями определения размера, итерирования, сортировки и поиска элементов. Стандартная библиотека предоставляет алгоритмы для решения большинства из наиболее распространенных задач, необходимых пользователям контейнеров.

Данный раздел резюмирует стандартные алгоритмы; в нем приводится несколько примеров их использования, объясняются ключевые принципы и методы реализации алгоритмов на языке C++, а также подробно рассматривается ряд важнейших алгоритмов.

Функциональные объекты (function objects) обеспечивают механизм, с помощью которого пользователь может приспособить стандартный алгоритм под свои нужды. Функциональные объекты предоставляют контейнеру информацию, необходимую для работы с пользовательскими данными. Поэтому в настоящем разде-

ле рассмотрение функциональных объектов фокусируется на том, как их можно создать и использовать.

18.2. Обзор алгоритмов стандартной библиотеки

На первый взгляд может показаться, что в стандартной библиотеке алгоритмов несметное число. Однако ж, их всего 60. Я нередко встречал классы, содержащие большее число функций-членов. Многие алгоритмы имеют между собой много общего в интерфейсе и поведении, что облегчает их изучение и понимание. Как и в случае с общеязыковыми средствами, программист может использовать лишь те алгоритмы, которые ему понятны и действительно нужны. Нет никакой необходимости и заслуги в том, чтобы использовать в программе как можно больше алгоритмов, или заменять простые и очевидные алгоритмы на сложные и хитроумные. Помните — очень важно писать программы наиболее очевидным и простым способом, чтобы их потом можно было легко читать и понимать (это важно даже для автора программы). С другой стороны, перед тем, как проделать что-то с элементами контейнера, подумайте, нет ли готового стандартного алгоритма для решения этой задачи. Если универсальный алгоритм уже есть, то зачем заново изобретать колесо?

Каждый алгоритм задается функциональным шаблоном (§13.3) или набором функциональных шаблонов. Поэтому алгоритмы могут работать с последовательностями элементов разных типов. Те алгоритмы, которые возвращают итератор (§19.3), в качестве сигнала о неудаче возвращают как правило итератор, указывающий на элемент, расположенный за концом входной последовательности. Например:

```
void f(list<string>& ls)
{
  list<string>::const_iterator p = find(ls.begin(), ls.end(), "Fred");

  if(p == ls.end())
  {
    // не нашли "Fred"
  }
  else
  {
    // здесь p указывает на "Fred"
  }
}
```

Алгоритмы не выполняют контроля границ последовательностей (диапазонов) ни на входе, ни на выходе. Ошибки выхода за границы диапазонов должны предотвращаться иными способами (§18.3.1, §19.3). Когда алгоритм возвращает итератор, его тип тот же, что и у итераторов на входе алгоритма. В частности, аргументы алгоритма определяют, возвращается *константный* или *неконстантный итератор*. Например:

```
void f(list<int>& li, const list<string>& ls)
{
  list<int>::iterator p = find(li.begin(), li.end(), 42);
  list<string>::const_iterator q = find(ls.begin(), ls.end(), "Ring");
}
```

Алгоритмы стандартной библиотеки реализуют множество полезных стандартных операций над элементами контейнеров, таких как просмотр, сортировка, поиск, вставка и удаление. Стандартные алгоритмы объявлены в пространстве имен ***std*** и сосредоточены в заголовочном файле <***algorithm***>. Большинство распространенных алгоритмов столь просты, что соответствующие им шаблонные функции реализуются как встраиваемые, и циклы с применением алгоритмов сильно при этом выигрывают с точки зрения быстродействия.

Стандартные функциональные объекты также объявляются в пространстве имен ***std***, но их определения сосредоточены в заголовочном файле <***functional***>. Они также легко поддаются встраиванию.

Немодифицирующие (неизменяющие значений и порядка следования элементов) алгоритмы используются для извлечения информации из последовательности элементов или для определения позиций элементов в этой последовательности (интервале):

Немодифицирующие алгоритмы (§18.5) <algorithm>	
for_each ( )	Выполняет операцию для каждого элемента последовательности.
find ( )	Находит первое вхождение значения в последовательности.
find_if ( )	Находит первое соответствие предикату в последовательности.
find_first_of	Находит значение одной последовательности в другой последовательности.
adjacent_find ( )	Находит пару смежных значений.
count ( )	Подсчитывает число вхождений значения в последовательности.
count_if ( )	Подсчитывает число соответствий предикату в последовательности.
mismatch ( )	Находит первые элементы, для которых последовательности отличаются.
equal ( )	Возвращает *true*, если последовательности попарно эквивалентны.
search ( )	Находит первое вхождение подпоследовательности в последовательность.
find_end ( )	Находит последнее вхождение подпоследовательности в последовательность.
search_n ( )	Находит *n-ое* вхождение значения в последовательность.

Большинство алгоритмов предоставляют пользователю возможность определить фактическую операцию, выполняемую над каждым элементом или парой элементов. Это делает стандартные алгоритмы гораздо более универсальными и полезными, чем они кажутся на первый взгляд. В частности, пользователь может сам указать критерии эквивалентности и различия элементов (§18.4.2). Где это уместно, наиболее очевидные операции предоставляются по умолчанию.

Модифицирующие алгоритмы имеют между собой мало общего, кроме очевидного факта, что они могут изменять значения элементов последовательности:

Модифицирующие алгоритмы (§18.6) <algorithm>	
transform ()	Применяет операцию к каждому элементу последовательности.
copy ()	Копирует последовательность, начиная с первого элемента.
copy_backward ()	Копирует последовательность, начиная с ее последнего элемента.
swap ()	Меняет местами два элемента.
iter_swap ()	Меняет местами два элемента, указуемые итераторами.
swap_ranges ()	Меняет местами элементы двух последовательностей.
replace ()	Заменяет элементы с указанным значением.
replace_if ()	Заменяет элементы, удовлетворяющие предикату.
replace_copy (())	Копирует последовательность, заменяя элементы с указанным значением.
replace_copy_if ()	Копирует последовательность, заменяя элементы, удовлетворяющие предикату.
fill ()	Заменяет все элементы заданным значением.
fill_n ()	Заменяет первые *n* элементов заданным значением.
generate ()	Заменяет все элементы результатом операции.
generate_n ()	Заменяет первые *n* элементов результатом операции.
remove ()	Удаляет элементы с указанным значением.
remove_if ()	Удаляет элементы, удовлетворяющие предикату.
remove_copy ()	Копирует последовательность, удаляя элементы с указанным значением.
remove_copy_if ()	Копирует последовательность, удаляя элементы, удовлетворяющие предикату.
unique ()	Удаляет равные смежные элементы.
unique_copy ()	Копирует последовательность, удаляя равные смежные элементы.
reverse ()	Меняет порядок следования элементов на обратный.
reverse_copy ()	Копирует последовательность в обратном порядке.
rotate ()	Циклически перемещает элементы.
rotate_copy ()	Копирует элементы в циклической последовательности.
random_shuffle ()	Перемещает элементы последовательности в случайном порядке.

Всякий хороший программный проект несет на себе индивидуальные черты проектировщиков и их интересы. Контейнеры и стандартные алгоритмы явно отражают намерение предоставить базовую основу для классических структур данных и алгоритмов работы с ними. Стандартная библиотека не только предоставляет минимум необходимых каждому программисту контейнеров и алгоритмов, но и содержит инструменты, необходимые для поддержки стандартных алгоритмов и для расширения библиотеки за пределы этого минимума.

В настоящем разделе упор делается не на проектировании алгоритмов, и даже не на их использовании (за исключением очевидных и простейших случаев). По поводу этих вопросов лучше обратиться к другой литературе (например, [Knuth, 1968] или [Tarjan, 1983]). Мы же просто перечисляем алгоритмы стандартной библиотеки и объясняем, как они реализуются на языке C++. Это позволит разобравшемуся в алгоритмах программисту правильно использовать стандартную библиотеку и расширять ее на основе принципов, на которых она сама и построена.

Стандартная библиотека предоставляет множество вариантов сортировки, поиска и иных путей манипулирования упорядоченными последовательностями:

Сортированные последовательности (§18.7) <algorithm>	
sort ( )	Сортировка с хорошей средней эффективностью.
stable_sort ( )	Сортировка с сохранением порядка эквивалентных элементов.
partial_sort ( )	Упорядочивает первую часть последовательности.
partial_sort_copy ( )	Копирует, упорядочивая первую часть результата.
nth_element ( )	Ставит *n*-ый элемент на нужное место.
lower_bound (( ))	Находит первое вхождение значения.
uper_bound (( ))	Находит первый элемент, больший заданного значения.
equal_range ( )	Находит подпоследовательность элементов с заданным значением.
binary_search ( )	Имеется ли данное значение в сортированной последовательности?
merge ( )	Слияние двух сортированных последовательностей.
inplace_merge ( )	Слияние двух последовательно отсортированных последовательностей.
partition ( )	Перемещает вперед элементы, удовлетворяющие предикату.
stable_partition ( )	Перемещает вперед элементы, удовлетворяющие предикату, сохраняя их относительный порядок следования.

Алгоритмы для работы со множествами (§18.7.5) <algorithm>	
includes ( )	Возвращает *true*, если последовательность является подпоследовательностью другой последовательности.
set_union ( )	Конструирует сортированное объединение.
set_intersection ( )	Конструирует сортированное пересечение.
set_difference ( )	Конструирует сортированную последовательность элементов, входящих в первую, но не во вторую последовательность.
set_symmetric_difference ( )	Конструирует сортированную последовательность элементов, входящих в одну из двух последовательностей.

Операции с кучей поддерживают последовательность в состоянии, облегчающей сортировку в случае возникновения такой необходимости:

Операции с кучей (§18.8) <algorithm>	
make_heap ()	Подготавливает последовательность к использованию в качестве кучи.
push_heap ()	Добавляет элемент к куче.
pop_heap ()	Удаляет элемент из кучи.
sort_heap ()	Сортирует кучу.

В библиотеке имеется ряд алгоритмов для выбора элементов, основанных на сравнении:

Минимум и максимум (§18.9) <algorithm>	
min ()	Меньшее из двух значений.
max ()	Большее из двух значений.
min_element ()	Наименьшее значение в последовательности.
max_element ()	Наибольшее значение в последовательности.
lexicographical_compare ()	Лексикографически первая из двух последовательностей.

Наконец, библиотека позволяет осуществлять перестановки элементов последовательности:

Перестановки (§18.10) <algorithm>	
next_permutation ()	Следующая перестановка в лексикографическом порядке.
prev_permutation ()	Предыдущая перестановка в лексикографическом порядке.

Кроме того, несколько обобщенных численных алгоритмов определены в заголовочном файле <***numeric***> (§22.6).

В описании алгоритмов имена параметров шаблона важны. ***In***, ***Out***, ***For***, ***Bi*** и ***Ran*** означают, соответственно: итератор ввода, итератор вывода, прямой итератор, двунаправленный итератор и итератор произвольного доступа (§19.2.1). ***Pred*** означает унарный предикат, а ***BinPred*** — бинарный предикат (§18.4.2), ***Cmp*** обозначает функцию сравнения (§17.1.4.1, §18.7.1), Op обозначает унарную операцию, а ***BinOp*** — бинарную операцию (§18.4). Чаще всего для имен параметров шаблонов применяют гораздо более длинные имена. Однако я считаю, что уже после беглого ознакомления со стандартной библиотекой применение длинных имен ухудшает, а не улучшает читаемость.

Итераторы произвольного доступа (*random-access iterators*) могут использоваться как *двунаправленные итераторы* (*bidirectional iterators*), двунаправленные итераторы — как *прямые итераторы* (*forward iterators*), а прямые итераторы — как *итераторы ввода* (*input iterators*) или *итераторы вывода* (*output iterators*) (см. §19.2.1). Переда-

ча шаблону типа, не обеспечивающего требуемых операций, приводит к ошибке во время конкретизации шаблона (§C.13.7). А в случае передачи типа, обеспечивающего требуемые операции с иной семантикой, поведение программы во время ее выполнения непредсказуемо (§17.1.4).

18.3. Диапазоны (интервалы) и контейнеры

Известен хороший универсальный принцип: то, что требуется часто, должно быть самым простым, легким в использовании и безопасным. Стандартная библиотека нарушает этот принцип ради универсальности. Для нее универсальность важнее всего. Например, мы можем найти два первых вхождения в список числа *42* следующим образом:

```
void f(list<int>& li)
{
 list<int>::iterator p = find(li.begin(), li.end(), 42); // первое вхождение
 if(p != li.end())
 {
  list<int>::iterator q = find(++p, li.end(), 42);       // второе вхождение
  // ...
 }
 // ...
}
```

Если бы *find*() выполняла операцию над контейнером, нам потребовался бы какой-нибудь дополнительный механизм для поиска второго вхождения. Очень важно, что выполнить обобщение такого «дополнительного механизма» на все контейнеры и алгоритмы не так-то просто. Вместо этого алгоритмы стандартной библиотеки работают над последовательностями элементов. То есть на вход алгоритма подается последовательность элементов, определяемая парой итераторов. Первый итератор ссылается на первый элемент последовательности, а второй — на место, расположенное сразу за последним элементом последовательности (§3.8, §19.2). Эта последовательность является полуоткрытой, поскольку она включает в себя первый из упомянутых только что элементов, но не второй из них. Такие полуоткрытые последовательности позволяют выразить бóльшинство алгоритмов без выделения пустой последовательность в отдельный случай.

Последовательности элементов, особенно последовательности с произвольным доступом, принято называть *интервалами* или *диапазонами* (*range*). Традиционной математической записью полуоткрытых интервалов служит [*first*, *last*) или [*first*, *last*[. Важно, что интервалы могут включать все элементы контейнера, или подмножество элементов контейнера. Некоторые последовательности, такие как, например, потоки I/O, не имеют отношения к контейнерам. Однако алгоритмы, выраженные в терминах последовательностей (интервалов) прекрасно работают и в этом случае.

18.3.1. Входные диапазоны

Использование *x.begin()*, *x.end()* для записи всех элементов x столь широко распространено, что даже часто утомляет и может привести к ошибке. Например, когда в коде имеется сразу несколько итераторов, можно совершить ошибку, предоставив алгоритму не ту пару итераторов (которые не образуют последовательность):

```
void f(list<string>& fruit, list<string>& citrus)
{
  typedef list<string>::const_iterator LI;

  LI p1 = find(fruit.begin(), citrus.end(), "apple");   // wrong!(разные последовательности)
  LI p2 = find(fruit.begin(), fruit.end(), "apple");    // ok
  LI p3 = find(citrus.begin(), citrus.end(), "pear");   // ok
  LI p4 = find(p2, p3, "peach");                        // wrong!(разные последовательности)
}
```

В данном примере две ошибки. Первая вполне очевидна (если подозреваешь, что она есть), но компилятору обнаружить ее нелегко. Вторую ошибку трудно обнаружить в реальной программе даже опытному программисту. Но можно упростить задачу, сократив число явно используемых итераторов. Сейчас я очерчу подход к решению проблемы, основанный на явном введении входных последовательностей. В остальной же части данной главы я не буду использовать явных входных последовательностей, чтобы удержать разговор о стандартных алгоритмах строго в рамках стандартной библиотеки.

Ключевая идея подхода заключается в явном формировании входной последовательности. Например:

```
template<class In, class T> In find(In first, In last, const T& v) // стандарт
{
  while(first!=last && *first!=v) ++first;
  return first;
}

template<class In, class T> In find(Iseq<In> r, const T& v)      // расширение
{
  return find(r.first, r.second, v);
}
```

При использовании аргумента *Iseq* перегрузка (§13.3.2) позволяет отдать предпочтение версии со входной последовательностью.

Естественно, входная последовательность формируется как пара (§17.4.1.2) итераторов:

```
template<class In> struct Iseq: public pair<In,In>
{
  Iseq(In i1, In i2) : pair<In,In>(i1,i2) { }
};
```

Мы могли бы и явно сформировать входную последовательность *Iseq*, необходимую для вызова второй версии алгоритма *find()*:

```
LI p= find(Iseq<LI>(fruit.begin(),fruit.end()), "apple");
```

Однако это еще более утомительно, чем вызывать оригинальную версию ***find*()**. На помощь приходит простая вспомогательная функция. Обычно, входная последовательность формируется на всех элементах контейнера (от ***begin*()** до ***end*()**):

```
template<class C> Iseq<typename C::iterator> iseq(C& c)   // для всего контейнера
{
  return Iseq<typename C::iterator>(c.begin(),c.end());
}
```

Теперь можно применять алгоритм к контейнеру в очень компактной форме:

```
void f(list<string>& ls)
{
  list<string>::iterator p = find(ls.begin(),ls.end(),"standard");
  list<string>::iterator q = find(iseq(ls),"extension");
  // ...
}
```

Легко определить вспомогательные функции ***iseq*()**, формирующие входные последовательности ***Iseq*** для массивов, потоков ввода и т.д. (§18.13[6]).

Введение ***Iseq*** позволяет явно отразить понятие входной последовательности, а вспомогательная функция ***iseq*()** позволяет устранить чреватое ошибками монотонное и утомительное дублирование кода при создании входных последовательностей через пару итераторов.

Понятие выходной последовательности также имеет некоторую ценность, но оно сложнее и его не так-то просто применить с пользой в реальном коде (§18.13[7]; см. также §19.2.4).

18.4. Классы функциональных объектов

Многие алгоритмы работают с последовательностями исключительно через итераторы и значения. Например, мы можем найти первый элемент последовательности со значением *7* следующим образом:

```
void f(list<int>& c)
{
  list<int>::iterator p = find(c.begin(),c.end(),7);
  // ...
}
```

В качестве более интересного примера мы можем заставить алгоритм выполнить код, который мы ему передаем (§3.8.4). Например, можно находить первый элемент последовательности со значением, меньшим *7*:

```
bool less_than_7(int v)
{
  return v<7;
}

void f(list<int>& c)
{
  list<int>::iterator p = find_if(c.begin(),c.end(),less_than_7);
  // ...
}
```

Функции, передаваемые в качестве аргумента, могут быть: логическими предикатами, арифметическими операциями, операциями извлечения информации из элементов и т.п. Но писать отдельные функции для каждого случая и неудобно, и неэффективно. К тому же, не существует функции, способной отразить все необходимые нам логические нюансы. Например, часто нужно вызывать функцию для каждого элемента так, чтобы она сохраняла информацию между отдельными вызовами и возвращала некоторый совокупный для всех ее вызовов результат. Здесь лучше подойдет функция-член класса (чем глобальная функция), поскольку объект этого класса может сохранять любую информацию. Кроме того, класс может обеспечить операции инициализации и извлечения такой информации.

Рассмотрим класс, объекты которого ведут себя как функции, призванный вычислять сумму:

```
template<class T> class Sum
{
  T res;

public:
  Sum (T i = 0) : res(i) { }          // инициализация
  void operator() (T x) {res += x; }  // накопление
  T result() const {return res; }     // возврат суммы
};
```

Ясно, что класс *Sum* предназначен для арифметических типов, для которых инициализация нулем и операция += определены. Например:

```
void f(list<double>& ld)
{
  Sum<double> s;

  s = for_each(ld.begin(), ld.end(), s);   // вызываем s() для каждого элемента ld
  cout<< "the sum is" << s.result() << '\n';
}
```

Здесь *for_each*() (§18.5.1) вызывает *Sum<double>::operator*() (*double*) для каждого элемента из *ld*, и возвращает объект, переданный в качестве третьего параметра.

Ключевая причина такого поведения состоит в том. что алгоритм *for_each*() не считает, что третий параметр является именно функцией. Вместо этого он полагает, что третий аргумент есть нечто, к чему можно применить операцию вызова и передать при этом необходимый аргумент. Определенный подходящим образом объект подойдет здесь так же, как и функция, и даже лучше. Например, перегруженная в классе операция () поддается встраиванию лучше, чем функция, переданная по указателю на функцию — как следствие выигрывает производительность кода.

Объекты классов, перегружающие операцию вызова (операцию ()), называются *объектами-функциями* или *функциональными объектами* (*function-like objects* или *functions objects*), а также *функторами* (*functors*).

18.4.1. Базовые классы функциональных объектов

Стандартная библиотека предоставляет множество полезных объектов-функций. Для помощи в написании классов функциональных объектов в заголовочном файле <*functional*> определены два базовых класса:

```
template<class Arg, class Res> struct unary_function
{
  typedef Arg argument_type;
  typedef Res result_type;
};

template<class Arg, class Arg2, class Res> struct binary_function
{
  typedef Arg first_argument_type;
  typedef Arg2 second_argument_type;
  typedef Res result_type;
};
```

Предназначение этих классов — дать стандартные имена типам аргументов и возвращаемых значений для использования в классах, производных от ***unary_function*** и ***binary_function***. Применяя эти базовые классы так, как это делает стандартная библиотека, программист избавит себя от необходимости осознания всех тонкостей, которые делают эти классы полезными (§18.4.4.1).

18.4.2. Предикаты

Предикат является функциональным объектом или просто функцией, имеющим возврат типа ***bool***. Например, в файле ***<functional>*** определяются:

```
template<class T> struct logical_not: public unary_function<T,bool>
{
  bool operator() (const T& x) const {return !x; }
};

template<class T> struct less: public binary_function<T,T,bool>
{
  bool operator() (const T& x, const T& y) const {return x<y; }
};
```

Унарные и бинарные предикаты с успехом используются в комбинации с алгоритмами. Например, мы можем сравнить две последовательности с целью найти первый элемент одной из них, который не меньше соответствующего элемента второй последовательности:

```
void f(vector<int>& vi, list<int>& li)
{
  typedef list<int>::iterator LI;
  typedef vector<int>::iterator VI;

  pair<VI,LI> p1 = mismatch(vi.begin(),vi.end(),li.begin(),less<int>());
  // ...
}
```

Алгоритм ***mismatch***() многократно применяет свой бинарный предикат к парам соответствующих элементов, пока не встретится нарушение условия (§18.5.4). Затем он возвращает итераторы на элементы, не удовлетворяющие поставленному условию. Поскольку *здесь нужен объект*, а *не тип*, используется ***less<int>***() (с *круглыми скобками*), а вовсе не ***less<int>***.

Вместо нахождения первого элемента, который «не меньше» соответствующего ему элемента другой последовательности, мы могли бы отыскивать элемент, который удовлетворяет условию «меньше». Для этого либо используем комплиментарный предикат ***greater_equal***:

```
p1 = mismatch(vi.begin(), vi.end(), li.begin(), greater_equal<int>());
```

либо меняем местами последовательности и применяем предикат ***less_equal***:

```
pair<LI, VI> p2 = mismatch(li.begin(), li.end(), vi.begin(), less_equal<int>());
```

В §18.4.4.4 я покажу, как реализовать предикат «не меньше».

18.4.2.1. Обзор предикатов

В заголовочном файле <***functional***> стандартная библиотека определяет несколько общеупотребительных предикатов:

Предикаты <functional>		
equal_to	Binary	arg1==arg2
not_equal_to	Binary	arg1!=arg2
greater	Binary	arg1>arg2
less	Binary	arg1<arg2
greater_equal	Binary	arg1>=arg2
less_equal	Binary	arg1<=arg2
logical_and	Binary	arg1&&arg2
logical_or	Binary	arg1||arg2
logical_not	Unary	!arg

Определения предикатов ***less*** и ***logical_not*** представлены в §18.4.2.

В дополнение к стандартным библиотечным предикатам пользователь может писать свои собственные. Пользовательские предикаты позволяют просто и изящно использовать контейнеры и алгоритмы стандартной библиотеки. В частности, пользовательские предикаты нужны для того, чтобы применить стандартные алгоритмы для нестандартных пользовательских классов. Например, рассмотрим вариант класса ***Club*** из §10.4.6:

```
class Person { /* ... */ };

struct Club
{
  string name;
  list<Person*> members;
  list<Person*> officers;
  // ...
  Club(const string& n);
};
```

Поиск конкретного клуба с заданным именем (значение поля name) в списке *list<Club>* задача вполне осмысленная. К сожалению, стандартный алгоритм *find_if()* ничего не знает о пользовательском типе *Club*. Библиотечные алгоритмы умеют выполнять тест на эквивалентность, но нам не нужно сравнивать клубы по всем статьям. Мы просто хотим использовать *Club::name* в качестве ключа поиска. Для этого мы напишем следующий предикат:

```
class Club_eq: public unary_function<Club, bool>
{
  string s;

public:
  explicit Club_eq(const string& ss) : s(ss) {}
  bool operator() (const Club& c) const {return c.name==s; }
};
```

Определять полезные предикаты несложно. После того как мы определим предикаты для своих собственных типов, использовать стандартные алгоритмы становится столь же просто и эффективно, как и для контейнеров с элементами простых типов. Например:

```
void f(list<Club>& lc)
{
  typedef list<Club>::iterator LCI;
  LCI p = find_if(lc.begin(), lc.end(), Club_eq("Dining Philosophers"));
  //...
}
```

18.4.3. «Арифметические» функциональные объекты.

При работе с числовыми классами полезно представить стандартные арифметические функции в виде функциональных объектов (объектов-функций). Для этого в файле *<functional>* стандартная библиотека определяет следующие операции:

Арифметические операции <functional>		
plus	Binary	arg1+arg2
minus	Binary	arg1 – arg2
multiplies	Binary	arg1*arg2
divides	Binary	arg1/arg2
modulus	Binary	arg1%arg2
negate	Unary	–arg

Мы могли бы при помощи *multiplies* перемножить элементы двух векторов (получив в результате третий вектор):

```
void discount(vector<double>& a, vector<double>& b, vector<double>& res)
{
transform(a.begin(), a.end(), b.begin(), back_inserter(res), multiplies<double>());
}
```

Функциональный шаблон **back_inserter**() описывается в §19.2.4. Несколько численных алгоритмов можно найти в §22.6.

18.4.4. Адаптеры («связывающие», для адаптирования функций-членов и указателей на функции, «отрицающие»)

Мы можем применять как собственные предикаты и арифметические функциональные объекты, так и те, что предоставляются стандартной библиотекой. Часто встречаются ситуации, когда необходимый предикат лишь незначительно отличается от уже существующих. Стандартная библиотека предоставляет для таких случаев ряд классов, позволяющих осуществлять композицию (адаптацию) объектов-функций:

- §18.4.4.1 Здесь рассматриваются классы, которые позволяют использовать двухаргументные (бинарные) функциональные объекты как одноаргументные (унарные) путем связывания (binding) одного из аргументов со значением.
- §18.4.4.2 Здесь рассматриваются классы, которые выступают как адаптеры функций-членов, позволяя использовать их в качестве аргументов алгоритмов.
- §18.4.4.3 Здесь рассматриваются классы, которые адаптируют указатели на функции для их использования в качестве аргументов алгоритмов.
- §18.4.4.4 Здесь рассматриваются классы, позволяющие придать предикату противоположный смысл.

В совокупности эти классы часто именуются *адаптерами* (*adapters*)[1]. Все они имеют общую структуру, опирающуюся на базовые классы **unary_function** и **binary_function** (§18.4.1). Для каждого из этих адаптеров предусмотрена вспомогательная функция, которая принимает исходный функциональный объект в качестве аргумента и возвращает требуемый функциональный объект. Последний с помощью метода **operator**()() выполняет необходимое действие. Можно сказать, что эти адаптеры реализуют функции высшего порядка, которые принимают на входе некоторые функции и порождают из них другие:

«Связывающие» и «отрицающие» адаптеры <functional>		
bind2nd(*y*)	***bind2nd***	Вызывает бинарную функцию с y в качестве второго аргумента.
bind1st(*x*)	***bind1st***	Вызывает бинарную функцию с x в качестве первого аргумента.
mem_fun()	***mem_fun_t***	Вызывает 0-аргументную функцию-член через указатель.
	mem_fun1_t	Вызывает унарную функцию-член через указатель.
	const_mem_fun_t	Вызывает 0-аргументную константную функцию-член через указатель.
	const_mem_fun1_t	Вызывает унарную константную функцию-член через указатель.

[1] Более точно — предикатные адаптеры или адаптеры предикатов. — *Прим. ред.*

«Связывающие» и «отрицающие» адаптеры <functional>		
mem_fun_ref()	*mem_fun_ref_t*	Вызывает 0-аргументную функцию-член через ссылку.
	mem_fun1_ref_t	Вызывает унарную функцию-член через ссылку.
	const_mem_fun_ref_t	Вызывает 0-аргументную константную функцию-член через ссылку.
	const_mem_fun1_ref_t	Вызывает унарную константную функцию-член через ссылку.
ptr_fun()	*pointer_to_unary_function*	Вызывает унарный указатель на функцию.
ptr_fun()	*pointer_to_binary_function*	Вызывает бинарный указатель на функцию.
not1()	*unary_negate*	Отрицает унарный предикат.
not2()	*binary_negate*	Отрицает бинарный предикат.

18.4.4.1. «Связывающие» адаптеры

Бинарные предикаты, такие как *less* (§18.4.2), полезны и достаточно гибки. Однако на практике чаще всего нужен предикат, который последовательно сравнивает заданное значение со значениями элементов контейнера. Функция *less_than_7*() (§18.4) является типичным примером. Предикат же *less* требует явного задания двух аргументов при каждом вызове, так что его нельзя применить в таких случаях непосредственно. Но мы можем определить шаблон *less_than*:

```
template<class T> class less_than : public unary_function<T, bool>
{
  T arg2;
public:
  explicit less_than (const T& x) : arg2 (x) { }
  bool operator () (const T& x) const {return x<arg2; }
};
```

и использовать его:

```
void f(list<int>& c)
{
  list<int>::const_iterator p = find_if(c.begin(), c.end(), less_than<int>(7));
  // ...
}
```

Мы обязаны писать *less_than<int>*(7), а не *less_than*(7), потому что шаблонный аргумент *<int>* не может быть выведен из типа аргумента конструктора *7* (§13.3.1).

Предикат *less_than* полезен во многих случаях. Заметим, что мы определили его, *связав* (зафиксировав) второй аргумент операции «меньше». Такая техника очень важна и широкоупотребительна, а стандартная библиотека предоставляет специальный класс и вспомогательную функцию для выполнения этой задачи:

```
template<class BinOp>
class binder2nd : public unary_function<typename BinOp::first_argument_type,
                                        typename BinOp::result_type>
```

```
{
protected:
  BinOp op;
  typename BinOp::second_argument_type arg2;

public:
  binder2nd(const BinOp& x, const typename BinOp::second_argument_type& v)
       : op(x), arg2(v) { }

  result_type operator() (const argument_type& x) const {return op(x,arg2); }
};

template<class BinOp, class T> binder2nd<BinOp> bind2nd(const BinOp& op, const T& v)
{
  return binder2nd<BinOp>(op,v);
}
```

Например, мы можем применить ***bind2nd***() для создания унарного предиката «меньше чем *7*» из бинарного предиката «меньше» и значения *7*.

```
void f(list<int>& c)
{
  list<int>::const_iterator p = find_if(c.begin(),c.end(),bind2nd(less<int>(),7));
  //...
}
```

Насколько это читаемо и эффективно? Даже в среднего качества реализации C++ это намного эффективнее по быстродействию и расходу памяти (легко допускает встраивание), чем версия с функцией ***less_than_****7*() из §18.4.

Эта система обозначений довольно логична, но к ней нужно привыкнуть. Часто бывает полезно определить явно именованную операцию со связанным аргументом:

```
template<class T> struct less_than: public binder2nd<less<T> >
{
  explicit less_than(const T& x) : binder2nd<less<T> >(less<T>(),x) { }
};

void f(list<int>& c)
{
  list<int>::const_iterator p = find_if(c.begin(),c.end(),less_than<int>(7));
  //...
}
```

Важно определять ***less_than*** именно через ***less***, а не через операцию <. При этом ***less_than*** воспользуется всеми преимуществами любой возможной специализации ***less*** (§13.5, §19.2.2).

Параллельно с ***bind2nd***() и ***binder2nd***, в файле <***functional***> определяются еще и ***bind1st***() и ***binder1st*** для связывания первого аргумента бинарных функций.

18.4.4.2. Адаптирование функций-членов

Большинство алгоритмов использует в своей работе стандартные или определенные пользователем операции. Естественно, возникает необходимость и в использовании функций-членов. Например (§3.8.5):

```
void draw_all (list<Shape*>& c)
{
  for_each (c.begin () , c.end () , &Shape: :draw) ;        // error
}
```

Но проблема заключается в том, что функция-член *mf*() должна вызываться от имени объекта: *p->mf*(), в то время как алгоритмы, например *for_each*(), вызывают свои аргументы-функции обычным способом: *f*(). Следовательно, мы нуждаемся в механизме создания чего-то такого, что позволило бы алгоритмам просто и эффективно вызывать функции-члены. Альтернативой было бы дублирование алгоритмов: по одной версии для функций-членов и по другой версии для обычных функций. И даже хуже того: потребовались бы разные версии для контейнеров объектов и для контейнеров указателей на объекты. Как и в ранее рассмотренном случае связывающих адаптеров (§18.4.4.1) проблема решается введением класса плюс вспомогательной функции. Рассмотрим сначала типичный случай, когда нужно вызвать функцию-член без аргумента для контейнеров указателей на объекты:

```
template<class R, class T> class mem_fun_t: public unary_function<T*,R>
{
  R (T: :*pmf) () ;
public:
  explicit mem_fun_t (R (T: :*p) () ) : pmf (p) { }
  R operator () (T* p) const {return (p->*pmf) () ; }    // вызов по указателю
};

template<class R, class T> mem_fun_t<R,T> mem_fun (R (T: :*f) () )
{
  return mem_fun_t<R,T> (f) ;
}
```

Это позволяет нам исправить пример с *Shape::draw*():

```
void draw_all (list<Shape*>& lsp)          // вызываем безаргументную функцию-член
                                           // через указатель на объект
{
  for_each (lsp.begin () , lsp.end () , mem_fun (&Shape: :draw) ) ;  // рисуем все фигуры
}
```

Теперь перейдем к функциям-членам, имеющим аргументы, для чего потребуется класс и соответствующая функция *mem_fun*(). Кроме того, потребуется еще и версия для работы от имени объектов (а не указателей) — ее называют *mem_fun_ref*(). Наконец, нам нужны версии для константных функций-членов:

```
template<class R, class T> mem_fun_t<R,T> mem_fun (R (T: :*f) () ) ;
// а также версии для унарных, константных и унарных константных функций-членов
// (см. таблицу в §18.4.4)
```

```
template<class R, class T> mem_fun_ref_t<R,T> mem_fun_ref(R (T::*f)());
// а также версии для унарных, константных и унарных константных функций-членов
// (см. таблицу в §18.4.4)
```

С этими адаптерами функций-членов, определенными в заголовочном файле <*functional*>, мы можем писать следующий клиентский код:

```
void f(list<string>& ls)    // используем для объекта безаргументную функцию-член
{
  typedef list<string>::iterator LSI;
  LSI p = find_if(ls.begin(),ls.end(),mem_fun_ref(&string::empty)); // находим ""
}

void rotate_all(list<Shape*>& ls, int angle)
// используем функцию-член, принимающую аргумент через указатель на объект
{
  for_each(ls.begin(),ls.end(),bind2nd(mem_fun(&Shape::rotate),angle));
}
```

В стандартной библиотеке нет необходимости адаптировать функции-члены с более чем одним аргументом, так как нет алгоритмов, принимающих функцию с более чем двумя аргументами.

18.4.4.3. Версии адаптеров для указателей на функции

Алгоритмам не нужно знать, является ли их аргумент-функция просто функцией, указателем на функцию или функциональным объектом. А вот «связывающим» адаптерам (§18.4.4.1) это знать нужно, так как они сохраняют копию для последующего применения. Поэтому в заголовочном файле <*functional*> имеются еще и версии адаптеров для указателей на функции, которые можно использовать со стандартными алгоритмами. Их определение и реализация весьма близки определению адаптеров функций-членов (§18.4.4.2) — снова используются пара функций и пара классов:

```
template<class A, class R> pointer_to_unary_function<A,R> ptr_fun(R (*f)(A));

template<class A, class A2, class R>
    pointer_to_binary_function<A,A2,R> ptr_fun(R (*f)(A, A2));
```

С этими адаптерами указателей на функции мы можем применить «связывающие» адаптеры следующим образом:

```
class Record { /* ... */ };

bool name_key_eq(const Record&, const char*);   // сравнение на базе имен
bool ssn_key_eq(const Record&, long);            // сравнение на базе чисел

void f(list<Record>& lr)                         // используем указатель на функцию
{
  typedef list<Record>::iterator LI;

  LI p = find_if(lr.begin(),lr.end(),bind2nd(ptr_fun(name_key_eq),"John Brown"));
  LI q = find_if(lr.begin(),lr.end(),bind2nd(ptr_fun(ssn_key_eq),1234567890));
  // ...
}
```

Тут мы ищем элементы списка ***lr***, которые соответствуют ключам поиска «***John Brown***» и ***1234567890***.

18.4.4.4. «Отрицатели»

Предикатные «отрицатели»[1] похожи на «связывающие» адаптеры тем, что принимают операцию и порождают на ее основе необходимую операцию. Определение и реализация «отрицателей» весьма близки определению адаптеров функций-членов (§18.4.4.2). В целом они тривиальны, но окончательной простоте мешают слишком длинные стандартные имена:

```
template<class Pred>
class unary_negate: public unary_function<typename Pred::argument_type, bool>
{
  Pred op;

public:
  explicit unary_negate(const Pred& p) : op(p) { }
  bool operator() (const argument_type& x) const {return !op(x); }
};

template<class Pred>
class binary_negate: public binary_function<typename Pred::first_argument_type,
                                            typename Pred::second_argument_type, bool>
{
  typedef first_argument_type Arg;
  typedef second_argument_type Arg2;

  Pred op;

public:
  explicit binary_negate(const Pred& p) : op(p) { }
  bool operator() (const Arg& x, const Arg2& y) const {return !op(x,y); }
};

template<class Pred> unary_negate<Pred> not1(const Pred& p);
template<class Pred> binary_negate<Pred> not2(const Pred& p);
```

Эти классы и функции объявлены в заголовочном файле *<functional>*. Имена *first_argument_type*, *second_argument_type* и т.д. берутся из стандартных базовых классов *unary_function и binary_function*.

Как и «связывающие адаптеры», «отрицатели» удобно применять с помощью их собственных вспомогательных функций. Например, мы можем построить бинарный предикат «не меньше чем» и применить его для поиска самой первой пары элементов, у которой первый элемент меньше второго элемента:

```
void f(vector<int>& vi, list<int>& li)     // пересмотренный пример из §18.4.2
{
  // ...
  p1 = mismatch(vi.begin(), vi.end(), li.begin(), not2(less<int>()));
  // ...
}
```

Здесь *p1* идентифицирует первую пару элементов, для которой предикат «не меньше чем» не выполняется.

---

[1] То есть, адаптеры, придающие противоположный смысл предикату. — *Прим. ред.*

Предикаты работают с логическими условиями, и поэтому здесь нет эквивалентов побитовых операций |, &, ^ и ~.

Естественно, разные типы адаптеров можно сочетать друг с другом. Например:

```
extern "C" int strcmp(const char*, const char*); // из <cstring>

void f(list<char*>& ls)                          // используется указатель на функцию
{
  typedef typename list<char*>::const_iterator LI;
  LI p= find_if(ls.begin(), ls.end(), not1(bind2nd(ptr_fun(strcmp), "funny")));
}
```

Здесь находится элемент списка *ls*, содержащий С-строку "*funny*". «Отрицатель» нужен из-за того, что *strcmp*() возвращает *нуль* при равенстве строк.

18.5. Немодифицирующие алгоритмы

Немодифицирующие алгоритмы являются основным средством поиска чего угодно в последовательности элементов без применения циклов. Кроме того, они позволяют получить информацию об элементах. Эти алгоритмы могут принимать константные итераторы (§19.2.1) и не должны (кроме алгоритма *for_each*()) вызывать операции, изменяющие элементы последовательности.

18.5.1. Алгоритм for_each()

Мы применяем библиотеки, чтобы воспользоваться ранее выполненной работой. Использование библиотечной функции, класса, алгоритма и т.д. экономит огромные усилия, которые в противном случае пришлось бы затратить на их проектирование, реализацию, и тестирование. Применение стандартной библиотеки делает код более понятным для любого человека, знакомого с этой библиотекой — ему не надо разбираться во внутреннем механизме чьей либо уникальной «программной кухни».

Одним из ключевых достоинств алгоритмов стандартной библиотеки является то, что они избавляют программиста от применения циклов. Явное программирование циклов и утомительно, и чревато ошибками. Алгоритм *for_each*() — простейший в том смысле, что он ничего не делает, кроме как устраняет необходимость в цикле. Фактически он просто вызывает свой аргумент-операцию для заданной последовательности:

```
template<class In, class Op> Op for_each(In first, In last, Op f)
{
  while (first!=last) f(*first++);
  return f;
}
```

Какие функции имеет смысл вызывать таким образом? Если вам нужно накапливать информацию от элементов, применяйте *accumulate*() (§22.6). Если нужно что-то найти в последовательности элементов — применяйте *find*() или *find_if*() (§18.5.2). Если вы изменяете или удаляете элементы, применяйте *replace*() (§18.6.4) или *remove*() (§18.6.5). Вообще, перед тем, как применить *for_each*(), подумайте, нет ли более специализированного алгоритма, применимого в вашем случае.

Возвращаемый алгоритмом *for_each*() результат есть либо функция, либо функциональный объект, переданный ему в качестве третьего параметра. Как показано в примере с классом *Sum* из §18.4, это позволяет вернуть информацию вызывающей функции.

Типичным применением алгоритма *for_each*() является задача об извлечении информации из элементов последовательности. Например, нам нужно собрать всю информацию об официальных лицах неизвестного числа клубов (структура *Club* из §18.4.2.1):

```
void extract(const list<Club>& lc, list<Person*>& off)
{
  for_each(lc.begin(),lc.end(),Extract_officers(off));
}
```

Параллельно примерам из §18.4 и §18.4.2 мы определяет класс функциональных объектов, предназначенный для извлечения необходимой информации. Необходимые имена из списка *list<Club>* переносим в наш список *list<Person*>*, то есть класс *Extract_officers* просто копирует сведения об официальных лицах из одного списка в другой:

```
class Extract_officers
{
  list<Person*>& lst;

public:
  explicit Extract_officers(list<Person*>& x) : lst(x) { }

  void operator()(const Club& c) const
  {copy(c.officers.begin(),c.officers.end(),back_inserter(lst));}
};
```

Наконец, мы можем распечатать полученный список официальных лиц, снова применив алгоритм *for_each*():

```
void extract_and_print(const list<Club>& lc)
{
  list<Person*> off;
  extract(lc,off);
  for_each(off.begin(),off.end(),Print_name(cout));
}
```

Написание *Print_name* мы оставляем в качестве упражнения (§18.13[4]).

Алгоритм *for_each*() классифицируется как немодифицирующий, потому что он сам по себе не изменяет последовательность. Однако будучи применен к неконстантной последовательности, этот алгоритм через свой третий аргумент может изменять элементы последовательности — см., например, использование *negate* в §11.9.

18.5.2. Алгоритмы поиска

Алгоритмы поиска просматривают элементы одной или двух последовательностей для нахождения значения или соответствия предикату. Простейший алгоритм *find*() из этого семейства алгоритмов просто ищет конкретное значение или производит поиск на соответствие предикату:

```
template<class In, class T> In find(In first, In last, const T& val);

template<class In, class Pred> In find_if(In first, In last, Pred p);
```

Алгоритмы *find*() и *find_if*() возвращают итератор для первого элемента, имеющего требуемое значение или отвечающего предикату, соответственно. Фактически, *find*() можно рассматривать как версию *find_if*() с предикатом ==. Почему бы тогда не назвать обе эти версии просто *find*()? Ответ заключается в том, что механизм разрешения перегрузки функций не всегда способен различить вызовы двух шаблонных функций с одинаковым числом аргументов. Рассмотрим пример:

```
bool pred(int);

void f(vector<bool(*f)(int)>& v1, vector<int>& v2)
{
  find(v1.begin(), v1.end(), pred);     // находим pred
  find_if(v2.begin(), v2.end(), pred);  // находим int для которого pred() возвращает true
}
```

Если бы *find*() и *find_if*() имели одно и то же имя, неожиданно возникла бы неоднозначность. В общем случае, суффикс *_if* означает, что алгоритм работает с предикатом.

Алгоритм *find_first_of*() находит первый элемент последовательности, имеющий соответствие во второй последовательности:

```
template<class For, class For2>
    For find_first_of(For first, For last, For2 first2, For2 last2);

template<class For, class For2, class BinPred>
    For find_first_of(For first, For last, For2 first2, For2 last2, BinPred p);
```

Например:

```
int x[] = { 1,3,4 };
int y[] = { 0,2,3,4,5};

void f()
{
  int* p = find_first_of(x,x+3,y,y+5);    // p = &x[1]
  int* q = find_first_of(p+1,x+3,y,y+5);  // q = &x[2]
}
```

Указатель p покажет на *x*[*1*], поскольку *3* — это первый элемент из *x*, имеющий значение, совпадающее со значением элемента из *y*. Аналогично, *q* покажет на *x*[*2*].

Алгоритм *adjacent_find*() найдет пару соседних совпадающих значений:

```
template<class For> For adjacent_find(For first, For last);
template<class For, class BinPred> For adjacent_find(For first, For last, BinPred p);
```

Возвращаемым значением является итератор на первый совпадающий элемент. Например:

```
void f(vector<string>& text)
{
  vector<string>::iterator p = adjacent_find(text.begin(), text.end());
```

```
    if (p!=text.end() && *p=="the")     // я снова продублировал "the"!
    {
      // ...
    }
}
```

18.5.3. Алгоритмы count() и count_if()

Алгоритмы *count*() и *count_if*() подсчитывают число вхождений в последовательность некоторого значения:

```
template<class In, class T>
  typename iterator_traits<In>::difference_type count(In first, In last, const T& val);

template<class In, class Pred>
  typename iterator_traits<In>::difference_type count_if(In first, In last, Pred p);
```

Интересен тип возврата алгоритма *count*(). Рассмотрим очевидную (и наивную) версию *count*():

```
template<class In, class T> int count(In first, In last, const T& val)
{
  int res = 0;

  while (first!=last)
    if (*first++ == val) ++res;

  return res;
}
```

Проблема состоит в том, что тип *int* не совсем подходит для типа возврата. На машинах с недостаточно большим *int* его может не хватить для хранения числа подсчитанных алгоритмом *count*() элементов последовательности. И наоборот, на машинах с большим *int* будет более эффективной реализация с возвратом *short*.

Ясно, что число элементов в последовательности не может быть больше максимальной разности между ее итераторами (§19.2.1). Поэтому, можно было бы определить тип возврата следующим образом:

```
typename In::difference_type
```

Однако стандартные алгоритмы должны работать и со встроенными массивами, а не только со стандартными контейнерами. Например:

```
void f(char* p, int size)
{
  int n = count(p,p+size,'e');    // подсчет числа вхождений буквы 'e'
}
```

К сожалению, *char*::difference_type* не проходит в C++. Проблема решается при помощи частичной специализации *iterator_traits* (§19.2.2).

18.5.4. Алгоритмы equal() и mismatch()

Алгоритмы ***equal***() и ***mismatch***() сравнивают две последовательности:

```
template<class In, class In2> bool equal(In first, In last, In2 first2);

template<class In, class In2, class BinPred>
bool equal(In first, In last, In2 first2, BinPred p);

template<class In, class In2> pair<In,In2> mismatch(In first, In last, In2 first2);

template<class In, class In2, class BinPred>
pair<In,In2> mismatch(In first, In last, In2 first2, BinPred p);
```

Алгоритм ***equal***() просто проверяет, эквивалентны ли все пары соответствующих элементов двух последовательностей; ***mismatch***() ищет первую пару неэквивалентных друг другу элементов, и возвращает итераторы на эти элементы. Конец второй последовательности не задается, то есть ***last2*** отсутствует. Вместо этого предполагается, что во второй последовательности элементов не меньше, чем в первой, так что вместо ***last2*** используется ***first2***+(***last-first***). Такой подход используется в рамках стандартной библиотеки в тех случаях, когда для операций над парами элементов задается пара последовательностей.

Как показано в §18.5.1, стандартные алгоритмы более полезны, чем это может показаться на первый взгляд, поскольку пользователи могут предоставлять предикаты, задающие критерии равенства и соответствия.

Заметьте, что последовательности не обязаны быть одного и того же типа. Например:

```
void f(list<int>& li, vector<double>& vd)
{
  bool b = equal(li.begin(),li.end(),vd.begin());
}
```

Все, что здесь требуется, это чтобы элементы последовательностей подходили в качестве операндов предикатов.

Две версии ***mismatch***() отличаются только использованием предикатов. Фактически, их можно реализовать одной функцией с умолчательным аргументом-шаблоном:

```
template<class In, class In2, class BinPred>
pair<In,In2> mismatch(In first, In last, In2 first2,
                      BinPred p = equal_to<typename In::value_type>()) //§18.4.2.1
{
  while( first!=last && p(*first, *first2) )
  {
    ++first;
    ++first2;
  }
  return pair<In,In2>(first,first2);
}
```

Разницу между двумя функциями и одной функцией с умолчательным аргументом легко выявляется любым, кто использует указатели на функции. Тем не менее, если мысленно представлять себе все варианты стандартных алгоритмов как версии

с предикатом по умолчанию, то число необходимых для запоминания вариантов сократится примерно вдвое.

18.5.5. Поисковые алгоритмы

Поисковые алгоритмы ***search***(), ***search_n***() и ***find_end***() выявляют вхождение некоторой последовательности в качестве подпоследовательности другой последовательности (для поиска одиночных элементов используйте ***find***() (§18.5.2) или ***binary_search***() (§18.7.2)):

```
template<class For, class For2>
For search (For first, For last, For2 first2, For2 last2) ;

template<class For, class For2, class BinPred>
For search (For first, For last, For2 first2, For2 last2, BinPred p) ;

template<class For, class For2>
For find_end (For first, For last, For2 first2, For2 last2) ;

template<class For, class For2, class BinPred>
For find_end (For first, For last, For2 first2, For2 last2, BinPred p) ;

template<class For, class Size, class T>
For search_n (For first, For last, Size n, const T& val) ;

template<class For, class Size, class T, class BinPred>
For search_n (For first, For last, Size n, const T& val, BinPred p) ;
```

Алгоритм ***search***() ищет свой второй аргумент-последовательность как подпоследовательность первого аргумента-последовательности. В случае успеха поиска возвращается итератор на первый совпадающий элемент в первой последовательности. Таким образом, возвращаемое значение всегда находится внутри [***first***, ***last***]. Конец последовательности ***last*** возвращается для указания на отрицательный результат поиска. Например:

```
string quote ("Why waste time learning, when ignorance is instantaneous?") ;

bool in_quote (const string& s)
{
  typedef string::const_iterator SCI;
  SCI p = search (quote.begin () , quote.end () , s.begin () , s.end () ) ; // ищем s в quote
  return p!=quote.end () ;
}

void g ()
{
  bool b1 = in_quote ("learning") ;       // b1 = true
  bool b2 = in_quote ("lemming") ;        // b2 = false
}
```

Алгоритм поиска подстроки ***search***() очень полезен, так как он универсален для любых последовательностей.

Умолчательным критерием совпадения элементов служит операция ==. Когда требуется иной критерий, применяется аргумент, являющийся бинарным предикатом (§18.4.2).

Алгоритм *find_end*() ищет свой второй аргумент-последовательность как подпоследовательность первого аргумента-последовательности. В случае обнаружения вхождения этот алгоритм возвращает итератор на последний совпадающий элемент первой последовательности. Другими словами, *find_end*() работает как *search*(), но с концевой фиксацией возвращаемого результата — он находит последнее (а не первое) вхождение второго аргумента в первый аргумент-последовательность.

Алгоритм *search_n*() находит последовательность из по крайней мере *n* вхождений аргумента-значения *val* в свой первый аргумент-последовательность. Он возвращает итератор на первый элемент найденной последовательности.

18.6. Модифицирующие алгоритмы

Если нужно изменить последовательность, ее можно итерировать и изменять значения элементов. Однако где это допустимо, мы стараемся уйти от такого стиля программирования в пользу более простых и регулярных приемов. Альтернативой является применение алгоритмов, которые проходят последовательность и выполняют требуемые действия. Когда мы просто читаем элементы последовательности, мы используем немодифицирующие алгоритмы (§18.5). Модифицирующие алгоритмы выполняют наиболее распространенные виды обновления данных. Причем некоторые алгоритмы действительно обновляют содержимое последовательности, в то время как другие строят новые последовательности на базе информации, извлеченной во время прохода по исходной последовательности.

Стандартные алгоритмы работают со структурами данных через итераторы. Это приводит к некоторым трудностям тогда, когда элемент вставляется в контейнер или удаляется из него. Действительно, имея один лишь итератор элемента, как мы найдем контейнер, из которого собственно и надо элемент удалить? Кроме как в случае специальных итераторов (например, в случае итераторов для вставки; §3.8 и §19.2.4), операции с итераторами не изменяют размера контейнеров. Вместо вставки и удаления элементов алгоритмы изменяют их значения, меняют элементы местами, копируют значения элементов. Даже алгоритм *remove*() работает посредством переписывания элементов, подлежащих удалению (§18.6.5). Вообще говоря, результаты наиболее фундаментальных модифицирующих алгоритмов являются модификациями их входов. Те же алгоритмы, которые, как представляется, изменяют последовательности, являются вариантами копирующих алгоритмов.

18.6.1. Копирующие алгоритмы

Копирование — это простейший способ получить из некоторой последовательности другую последовательность. Определения простейших копирующих алгоритмов тривиальны:

```
template<class In, class Out> Out copy(In first, In last, Out res)
{
  while(first!=last) *res++ = *first++;
  return res;
}
```

```
template<class Bi, class Bi2> Bi2 copy_backward(Bi first, Bi last, Bi2 res)
{
  while(first!=last) *--res = *--last;
  return res;
}
```

Результат копирующего алгоритма не обязан быть контейнером. Подойдет все, что можно описать выходным итератором (§19.2.6). Например:

```
void f(list<Club>& lc, ostream& os)
{
  copy(lc.begin(), lc.end(), ostream_iterator<Club>(os));
}
```

Для чтения последовательности нам требуется знать, откуда начинать читать и где заканчивать. Для записи же требуется лишь итератор, указывающий куда писать. Однако нужно соблюдать осторожность, чтобы не произвести запись за конец последовательности. Один из способов застраховаться от такой проблемы состоит в применении специальных итераторов вставки (§19.2.4), увеличивающих размеры последовательности по мере необходимости. Например:

```
void f(const vector<char>& vs, vector<char>& v)
{
  copy(vs.begin(), vs.end(), v.begin());          // может писать за конец v
  copy(vs.begin(), vs.end(), back_inserter(v)); // добавляет элементы из vs в конец v
}
```

Входная и выходная последовательности могут перекрываться. Мы используем ***copy***(), когда последовательности не перекрываются или когда конец выходной последовательности находится в пределах входной последовательности. Мы используем ***backward_copy***(), когда начало выходной последовательности находится внутри входной последовательности. Таким образом, никакие элементы не будут переписаны до того, как их значения будут скопированы. См. также §18.13[13].

Естественно, чтобы скопировать что-нибудь в направлении от конца к началу, нам нужен двунаправленный итератор (§19.2.1) как для входной, так и для выходной последовательности. Например:

```
void f(vector<char>& vc)
{
  copy_backward(vc.begin(), vc.end(), ostream_iterator<char>(cout)); // error

  vector<char> v(vc.size());
  copy_backward(vc.begin(), vc.end(), v.end());                        // ok
  copy(v.begin(), v.end(), ostream_iterator<char>(cout));              // ok
}
```

Часто нужно копировать лишь элементы, отвечающие некоторому критерию. К сожалению, ***copy_if***() как-то выпал из набора стандартных алгоритмов (mea culpa — моя вина (лат.)). С другой стороны, определить его совсем не трудно:

```
template<class In, class Out, class Pred>
Out copy_if(In first, In last, Out res, Pred p)
{
  while(first!=last)
```

```
    {
      if (p (*first) ) *res++ = *first;
      ++first;
    }
    return res;
}
```

Теперь, если мы, например, захотим распечатать лишь элементы, значение которых больше *n*, то это можно сделать следующим образом:

```
void f (list<int>& ld, int n, ostream& os)
{
  copy_if (ld.begin () , ld.end () , ostream_iterator<int> (os) , bind2nd (greater<int> () , n) ) ;
}
```

См. также *remove_copy_if*() (§18.6.5).

18.6.2. Алгоритм transform()

Хотя это и сбивает с толку, но *transform* () вовсе не обязательно изменяет входную последовательность. Вместо этого алгоритм работает над выходной последовательностью, формируя ее элементы в виде результата предоставляемой пользователем операции, выполняемой над элементами входной последовательности:

```
template<class In, class Out, class Op>
Out transform (In first, In last, Out res, Op op)
{
  while (first!=last) *res++ = op (*first++) ;
  return res;
}

template<class In, class In2, class Out, class BinOp>
Out transform (In first, In last, In2 first2, Out res, BinOp op)
{
  while (first!=last) *res++ = op (*first++, *first2++) ;
  return res;
}
```

Алгоритм *transform* (), читающий единственную входную последовательность для порождения значений элементов выходной последовательности, аналогичен алгоритму *copy* (). Но вместо копирования значений элементов, он записывает результат операции над элементами. Мы даже могли бы определить *copy*() как *transform* () с операцией, возвращающей свой аргумент:

```
template<class T> T identity (const T& x) {return x; }

template<class In, class Out> Out copy (In first, In last, Out res)
{
  return transform (first, last, res, identity<typename iterator_traits<In>: : value_type>) ;
}
```

Явная квалификация *identity* потребовалась для получения конкретной функции из функционального шаблона. Шаблон *iterator_traits* (§19.2.2) используется для именования типа элементов, на которые указывает *In*.

Под другим углом зрения ***transform*()** можно рассматривать как вариант алгоритма ***for_each*()**, явно формирующего свой выход. Например, вот как при помощи ***transform*()** порождается список имен клубов из списка объектов типа ***Club*** (§18.4.2.1):

```
string nameof(const Club& c) {return c.name; }

void f(list<Club>& lc)
{
  transform (lc.begin(), lc.end(), ostream_iterator<string>(cout, "\n"), nameof);
}
```

Одна из причин, по которой алгоритм назван ***transform*()**[1], заключается в том, что результат этой операции часто пишется туда, откуда пришел аргумент. Рассмотрим уничтожение объектов, адресуемых набором указателей:

```
struct Delete_ptr
{
  template<class T> T* operator() (T* p) const {delete p; return 0; }
};

void purge (deque<Shape*>& s)
{
  transform (s.begin(), s.end(), s.begin(), Delete_ptr());
}
```

Алгоритм ***transform*()** всегда трансформирует выходную последовательность. Здесь же я направляю результат назад во входную последовательность, так что ***Delete_ptr*() (*p*)** имеет тот же эффект, что и ***p*=*Delete_ptr*() (*p*)**. Поэтому я и решил возвращать ***0*** из ***Delete_ptr*::*operator*() ()**.

Вариант алгоритма ***transform*()**, имеющий дело с двумя последовательностями, позволяет комбинировать информацию из двух источников. Например, программа анимации может иметь процедуру, обновляющую состояние списка фигур путем применения операции трансляции (сдвига):

```
Shape* move_shape(Shape* s, Point p)            // *s += p
{
  s->move_to(s->center()+p);
  return s;
}

void update_positions (list<Shape*>& ls, vector<Point>& oper)
{
  // выполнить операцию над соответствующим объектом:
  transform (ls.begin(), ls.end(), oper.begin(), ls.begin(), move_shape);
}
```

В принципе, никакой необходимости в возврате функции ***move_shape*()** нет, но алгоритм ***transform*()** «настаивает» на том, чтобы возврат его операции присваивался чему-либо. Поэтому функция ***move_shape*()** возвращает свой первый аргумент, и его можно записать туда, откуда он был получен.

[1] Transform — изменять, преобразовывать, трансформировать. — *Прим. ред.*

Иногда такой возможности нет. Например, когда речь идет о чужой операции, не имеющей возврата (и которую нет желания или возможности исправлять). Или когда последовательность константная. В таком случае мы можем определить вариант *for_each*() с двумя последовательностями, соответствующий варианту *transform*() с двумя последовательностями:

```
template<class In, class In2, class BinOp>
BinOp for_each (In first, In last, In2 first2, BinOp op)
{
  while (first!=last) op (*first++, *first2++);
  return op;
}

void update_positions (list<Shape*>& ls, vector<Point>& oper)
{
  for_each (ls.begin(), ls.end(), oper.begin(), move_shape);
}
```

А бывают еще и случаи, когда полезно иметь выходной итератор, не выполняющий никакой фактической записи (§19.6[2]).

Среди стандартных алгоритмов отсутствуют алгоритмы, работающие с тремя и более последовательностями. Правда, написать такие алгоритмы несложно. А в качестве альтернативы можно многократно использовать *transform*().

18.6.3. Алгоритм unique()

При сборе информации на хранение возможно ее дублирование. Алгоритмы *unique*() и *unique_copy*() устраняют последовательные повторы значений:

```
template<class For> For unique (For first, For last);
template<class For, class BinPred> For unique (For first, For last, BinPred p);

template<class In, class Out> Out unique_copy (In first, In last, Out res);
template<class In, class Out, class BinPred>
Out unique_copy (In first, In last, Out res, BinPred p);
```

Алгоритм *unique*() устраняет последовательное дублирование значений, а *unique_copy*() не допускает его. Например:

```
void f (list<string>& ls, vector<string>& vs)
{
  ls.sort();          // сортировка списка (§17.2.2.1)
  unique_copy (ls.begin(), ls.end(), back_inserter (vs));
}
```

Здесь *ls* копируется в vs таким образом, что дубликаты автоматически удаляются. Применение *sort*() позволяет расположить одинаковые строки смежным образом.

Как и другие стандартные алгоритмы, *unique*() работает с итераторами. У него нет возможности знать тип контейнера, на элементы которого указывают итераторы, так что он не может модифицировать контейнер. Он только может изменять значения элементов. Поэтому он не удаляет одинаковые элементы, как мы наивно могли бы предположить. Вместо этого он последовательно помещает уникальные элементы в начало последовательности и возвращает итератор на конец подпоследовательности уникальных элементов:

```
template<class For> For unique (For first, For last)
{
  first = adjacent_find (first, last) ;   // §18.5.2
  return unique_copy (first, last, first) ;
}
```

Элементы за уникальной подпоследовательностью остаются неизменными. Поэтому нельзя сказать, что в векторе будут удалены все дубликаты:

```
void f(vector<string>& vs)              // внимание: плохой код!
{
  sort (vs.begin () , vs.end () ) ;     // сортировка вектора
  unique (vs.begin () , vs.end () ) ;   // устраняет дубликаты? Нет!
}
```

Фактически, перемещая для устранения дубликатов задние элементы последовательности вперед, алгоритм ***unique***() может произвести новые дубликаты. Например:

```
int main ()
{
  char v[] = "abbcccde";

  char* p = unique (v, v+strlen (v) ) ;
  cout<< v << ' ' << p-v << '\n' ;
}
```

порождает следующий результат:

```
abcdecde 5
```

Таким образом, ***p*** показывает на второе ***c***.

Алгоритмы, которые вроде как должны удалять элементы (но не делают этого) присутствуют в двух формах: простые версии, перестраивающие элементы как ***unique***(), и версии, создающие новую последовательность, как это делает ***unique_copy***(). Суффикс ***_copy*** используется для того, чтобы различать эти два вида алгоритмов.

Для устранения дубликатов из контейнера, его нужно явным образом «сжать» (уменьшить размер — shrink):

```
template<class C> void eliminate_duplicates (C& c)
{
  sort (c.begin () , c.end () ) ;                                  // сортируем
  typename C::iterator p = unique (c.begin () , c.end () ) ;       // сжимаем
  c.erase (p, c.end () ) ;                                         // укорачиваем
}
```

Отметим, что ***eliminate_duplicates***() не имеет смысла для встроенных массивов, в то время как алгоритм ***unique***() вполне допустим и в этом случае.

Пример с ***unique_copy***() приведен в §3.8.3.

18.6.3.1. Критерии сортировки

Чтобы удалить все дубликаты, входную последовательность нужно отсортировать (§18.7.1). Как ***unique***(), так и ***unique_copy***() по умолчанию используют операцию == для сравнения элементов, но пользователь может предоставить и собственную операцию. Например, можно изменить пример из §18.5.1 так, чтобы удалить

все одинаковые имена. После извлечения имен официальных лиц клубов (тип *Club*), мы располагаем списком off типа *list<Person*>* (§18.5.1). После этого можно удалить дубликаты следующим образом:

```
eliminate_duplicates(off);
```

Однако такой метод опирается на сортировку указателей и предполагает, что каждый указатель уникально идентифицирует официальное лицо клуба. В общем случае, нужно внимательнее анализировать записи типа *Person*, чтобы ответить на вопрос об их эквивалентности. Например, можно написать следующее:

```
bool operator==(const Person& x, const Person& y)   // эквивалентность объектов
{
  // сравниваем x и y на равенство
}

bool operator<(const Person& x, const Person& y)    // "меньше чем" для объектов
{
  // сравниваем x и y с точки зрения упорядочения
}

bool Person_eq(const Person* x, const Person* y)    // равенство через указатель
{
  return *x == *y;
}

bool Person_lt(const Person* x, const Person* y)    // "меньше" через указатель
{
  return *x < *y;
}

void extract_and_print(const list<Club>& lc)
{
  list<Person*> off;

  extract(lc, off);
  off.sort(Person_lt);

  list<Person*>::iterator p = unique(off.begin(), off.end(), Person_eq);
  for_each(off.begin(), p, Print_name(cout));
}
```

Здесь я применяю *list::sort()* (§17.2.2.1), поскольку стандартный алгоритм *sort()* (§18.7.1) требует итераторов произвольного доступа, а *list* предоставляет лишь двунаправленные итераторы (§17.1.2).

Полезно также убедиться, что критерий сравнения, применяемый при сортировке, соответствует критерию сравнения, применяемому при удалении дубликатов. Умолчательные операции == и < с указателями редко когда полезны для сравнения указуемых объектов.

18.6.4. Алгоритмы замены элементов

Алгоритмы замены проходят последовательность, заменяя некоторые значения на заданное значение. Они следуют модели алгоритмов *find*/*find_if* и *unique*/*unique_copy*, порождая таким образом четыре варианта. И снова код весьма прост и иллюстративен:

```
template<class For, class T>
void replace (For first, For last, const T& val, const T& new_val)
{
  while (first!=last)
  {
    if(*first == val) *first = new_val;
    ++first;
  }
}

template<class For, class Pred, class T>
void replace_if(For first, For last, Pred p, const T& new_val)
{
  while (first!=last)
  {
    if(p(*first)) *first = new_val;
    ++first;
  }
}

template<class In, class Out, class T>
Out replace_copy (In first, In last, Out res, const T& val, const T& new_val)
{
  while (first!=last)
  {
    *res++ = (*first == val) ? new_val : *first;
    ++first;
  }
  return res;
}

template<class In, class Out, class Pred, class T>
Out replace_copy_if(In first, In last, Out res, Pred p, const T& new_val)
{
  while (first!=last)
  {
    *res++ = p(*first) ? new_val : *first;
    ++first;
  }
  return res;
}
```

Вот пример, когда мы хотим пройтись по списку строк, заменяя английскую транслитерацию моего родного города ***Aarhus*** на его истинное название ***Århus***:

```
void f(list<string>& towns)
{
  replace(towns.begin(),towns.end(),"Aarhus"," Århus");
}
```

Здесь используется расширенный набор символов (§C.3.3).

18.6.5. Алгоритмы удаления элементов

Эти алгоритмы удаляют элементы последовательности, отталкиваясь от их значений или основываясь на предикате:

```
template<class For, class T> For remove(For first, For last, const T& val);

template<class For, class Pred> For remove_if(For first, For last, Pred p);

template<class In, class Out, class T>
    Out remove_copy(In first, In last, Out res, const T& val);

template<class In, class Out, class Pred>
    Out remove_copy_if(In first, In last, Out res, Pred p);
```

Предположив, что клубы (структура *Club* из §18.4.2.1) имеют адрес, мы могли бы создать список клубов из Копенгагена:

```
class located_in: public unary_function<Club,bool>
{
  string town;

public:
  located_in(const string& ss) :town(ss) { }
  bool operator()(const Club& c) const {return c.town == town;}
};

void f(list<Club>& lc)
{
  remove_copy_if(lc.begin(),lc.end(),
          ostream_iterator<Club>(cout),not1(located_in("København")));
}
```

Таким образом, алгоритм ***remove_copy_if***() — это фактически ***copy_if***() (§18.6.1) с инвертированным условием. То есть элемент помещается алгоритмом ***remove_copy_if***() в выходной поток, если он не удовлетворяет предикату.

«Простой» алгоритм ***remove***() записывает все несовпадающие элементы в начало последовательности и возвращает итератор на конец уплотненной подпоследовательности (см. также §18.6.3).

18.6.6. Алгоритмы fill() и generate()

Алгоритмы ***fill***() и ***generate***() предназначены для систематического присвоения последовательностям значений:

```
template<class For, class T> void fill(For first, For last, const T& val);

template<class Out, class Size, class T> void fill_n(Out res, Size n, const T& val);

template<class For, class Gen> void generate(For first, For last, Gen g);

template<class Out, class Size, class Gen> void generate_n(Out res, Size n, Gen g);
```

Алгоритм ***fill***() последовательно присваивает указанное значение; алгоритм ***generate***() присваивает значения, возвращаемые последовательными вызовами его аргумента-функции. Версии с суффиксом ***_n*** осуществляют присваивания лишь первым ***n*** элементам последовательности.

Например, используем генераторы случайных чисел ***Randint*** и ***Urand*** из §22.7:

```
int v1[900];
int v2[900];
vector v3;

void f()
{
  fill(v1, &v1[900], 99);
  generate(v2, &v2[900], Randint()); // присваиваем случайные значения (§22.7)

  // вывод 200 случайных чисел в интервале [0..99]:
  generate_n(ostream_iterator<int>(cout), 200, Urand(100)); //см. §22.7

  fill_n(back_inserter(v3), 20, 99);    // добавляем к v3 20 элементов со значением 99
}
```

Функции ***fill***() и ***generate***() осуществляют присваивание, а не инициализацию. Если вам требуется манипулировать неинициализированными областями памяти, чтобы, например, создать в них объекты заданного типа и состояния, используйте алгоритмы вроде ***uninitialized_fill***() из <***memory***> (§19.4.4).

18.6.7. Алгоритмы reverse() и rotate()

Иногда нужно изменить порядок элементов последовательности:

```
template<class Bi> void reverse(Bi first, Bi last);
template<class Bi, class Out> Out reverse_copy(Bi first, Bi last, Out res);

template<class For> void rotate(For first, For middle, For last);
template<class For, class Out> Out rotate_copy(For first, For middle, For last, Out res);

template<class Ran> void random_shuffle(Ran first, Ran last);
template<class Ran, class Gen> void random_shuffle(Ran first, Ran last, Gen& g);
```

Алгоритм ***reverse***() изменяет порядок элементов последовательности на обратный, так что первый элемент становится последним и т.д. Алгоритм ***reverse_copy***() производит копию входной последовательности, но с обратным порядком следования элементов.

Алгоритм ***rotate***() рассматривает последовательность [***first***, ***last***) как круговую и циклически сдвигает ее элементы до тех пор, пока средний элемент последовательности (указывается параметром ***middle***) не окажется на месте, на котором ранее располагался первый элемент. То есть элемент из некоторой позиции ***first***+***i*** передвигается в позицию ***first***+(***i***+(***last***-***middle***))%(***last***-***first***). Целочисленное деление % — вот что делает сдвиг циклическим, а не просто сдвигом влево. Например:

```
void f()
{
  string v[] = {"Frog", "and", "Peach"};
  reverse(v, v+3);              // Peach and Frog
  rotate(v, v+1, v+3);          // and Frog Peach
}
```

Алгоритм ***rotate_copy***() копирует элементы в выходную последовательность с циклическим сдвигом.

По умолчанию ***random_shuffle***() перетасовывает последовательность при помощи генератора случайных равномерно распределенных чисел. То есть алгоритм выполняет перестановки элементов последовательности таким образом, что любая перестановка имеет одинаковый шанс быть выбранной. Если вам нужно другое распределение или более качественный генератор случайных чисел, вы можете предоставить собственные. При вызове ***random_shuffle***(***b***, ***e***, ***r***) генератор вызывается с аргументом, равным количеству элементов последовательности, то есть вызов генератора ***r***(***b***–***e***) возвращает значение в диапазоне [***0***, ***e***–***b***). Если ***My_rand*** — это такой генератор, то мы можем перетасовать колоду карт следующим образом:

```
void f(deque<Card>& dc, My_rand& r)
{
  random_shuffle(dc.begin(),dc.end(),r);
  //...
}
```

Перемещение элементов алгоритмами ***rotate***() и др. реально выполняется с помощью функции ***swap***() (§18.6.8).

18.6.8. Обмен элементов последовательностей местами

Чтобы сделать с элементами контейнера что-нибудь более-менее интересное, нам нужно уметь обменивать элементы местами (перемещать элементы). Наиболее просто и эффективно такой обмен выполняется алгоритмами семейства "***swap***":

```
template<class T> void swap(T& a, T& b)
{
  T tmp = a;
  a = b;
  b = tmp;
}

template<class For, class For2> void iter_swap(For x, For2 y);

template<class For, class For2> For2 swap_ranges(For first, For last, For2 first2)
{
  while(first!=last) iter_swap(first++, first2++);
  return first2;
}
```

При обмене элементов местами приходится создавать временный объект. Существуют хитрые приемы, позволяющие избавиться от этого. Но ради простоты и очевидности их лучше не применять. Алгоритм ***swap***() имеет специализации для важных типов, для которых это имеет большое значение (§16.3.9, §13.5.2).

Алгоритм ***iter_swap***() обменивает местами элементы, указуемые аргументами-итераторами.

Алгоритм ***swap_ranges***() обменивает элементы двух входных последовательностей.

18.7. Сортировка последовательностей

Когда данные собраны, их часто следует отсортировать. Ведь после того, как последовательность отсортирована, возможности удобной работы с ней резко возрастают.

Для сортировки последовательностей нужны способы сравнения элементов. Это делается с помощью бинарных предикатов (§18.4.2). По умолчанию для сравнения элементов используется операция <.

18.7.1. Сортировка

Алгоритмы сортировки требуют итераторов произвольного доступа (§19.2.1). Таким образом, они лучше всего работают для контейнера ***vector*** (§16.3) и подобных ему:

```
template<class Ran> void sort(Ran first, Ran last);
template<class Ran, class Cmp> void sort(Ran first, Ran last, Cmp cmp);

template<class Ran> void stable_sort(Ran first, Ran last);
template<class Ran, class Cmp> void stable_sort(Ran first, Ran last, Cmp cmp);
```

Стандартный контейнер ***list*** (§17.2.2) не предоставляет итераторов произвольного доступа, так что списки следует сортировать их специфическими операциями (§17.2.2.1).

Базовый алгоритм ***sort***() весьма эффективен — в среднем как *N*****log***(*N*) — но в худшем случае как ***O***(*N***N*). По счастью, худшие случаи довольно редки. Если важно гарантировать приличное поведение алгоритма во всех случаях (включая худшие), то следует применить алгоритм ***stable_sort***() с эффективностью порядка *N*****log***(*N*)****log***(*N*), улучшающейся до *N*****log***(*N*) в случае наличия достаточного объема свободной памяти. Относительный порядок следования одинаковых элементов алгоритмом ***stable_sort***() сохраняется неизменным (алгоритмом ***sort***() — нет).

Иногда нужны лишь несколько первых элементов отсортированной последовательности. В таких случаях имеет смысл сортировать последовательность до момента получения нескольких первых элементов, расположенных упорядоченно (так называемая частичная сортировка — *partial sort*):

```
template<class Ran> void partial_sort(Ran first, Ran middle, Ran last);

template<class Ran, class Cmp>
void partial_sort(Ran first, Ran middle, Ran last, Cmp cmp);

template<class In, class Ran>
Ran partial_sort_copy(In first, In last, Ran first2, Ran last2);

template<class In, class Ran, class Cmp>
Ran partial_sort_copy(In first, In last, Ran first2, Ran last2, Cmp cmp);
```

«Простые» алгоритмы ***partial_sort***() расставляют по порядку элементы в диапазоне от ***first*** до ***middle***. Алгоритмы ***partial_sort_copy***() выдают *N* элементов, где *N* — это наименьшее из длин двух последовательностей (входной и выходной). Нужно указать и начало, и конец выходной последовательности, поскольку это определяет число элементов, подлежащих сортировке. Например:

```
class Compare_copies_sold
{
public:
  bool operator() (const Book& b1, const Book& b2) const
  {return b1.copies_sold()<b2.copies_sold(); }  // сортировка в убывающем порядке
};

void f(const vector<Book>& sales)               // находит 10 лучших книг
{
  vector<Book> bestsellers(10);

  partial_sort_copy(sales.begin(), sales.end(),
                    bestsellers.begin(), bestsellers.end(),
                    Compare_copies_sold());

  copy(bestsellers.begin(), bestsellers.end(), ostream_iterator<Book>(cout, "\n"));
}
```

Поскольку выходная последовательность в алгоритме *partial_sort_copy*() должна индицироваться итератором произвольного доступа, мы не можем произвести сортировку непосредственно в выходной поток *cout*.

Наконец, имеются алгоритмы, призванные выполнять сортировку до тех пор, пока *N-й* элемент не займет свое окончательное место (за этим элементом уже не будут располагаться элементы, меньше его величиной):

```
template<class Ran> void nth_element(Ran first, Ran nth, Ran last);
template<class Ran, class Cmp> void nth_element(Ran first, Ran nth, Ran last, Cmp cmp);
```

Этот алгоритм особенно полезен для экономистов, социологов, учителей и т.д., поскольку им часто нужны медианы распределений, процентные отношения и другая статистика.

18.7.2. Бинарный поиск

Последовательный поиск, такой, например, как *find*() (§18.5.2), ужасно неэффективен для длинных последовательностей, но это все, что мы можем сделать без сортировки и хэширования (§17.6). Но после того, как последовательность отсортирована, можно выполнять быстрый бинарный поиск:

```
template<class For, class T> bool binary_search(For first, For last, const T& val);

template<class For, class T, class Cmp>
bool binary_search(For first, For last, const T& value, Cmp cmp);
```

Например:

```
void f(list<int>& c)
{
  if(binary_search(c.begin(), c.end(), 7))      // имеется ли 7 в с?
  {
    // ...
  }
  // ...
}
```

Алгоритм ***binary_search*** () возвращает значение типа ***bool***, показывающее, найдено ли искомое значение в последовательности. Как и в случае с ***find*** (), мы часто хотим знать, где именно в последовательности расположено искомое значение. Однако в последовательности может оказаться несколько таких значений, и нам могут потребоваться либо первый из них, либо вообще все такие элементы. Как следствие, имеются алгоритмы для выявления диапазона одинаковых элементов — ***equal_range*** (), и для уточнения их нижних и верхних границ — ***lower_bound*** () и ***upper_bound*** ():

```
template<class For, class T> For lower_bound(For first, For last, const T& val);
template<class For, class T, class Cmp>
For lower_bound(For first, For last, const T& val, Cmp cmp);

template<class For, class T> For upper_bound(For first, For last, const T& val);
template<class For, class T, class Cmp>
For upper_bound(For first, For last, const T& val, Cmp cmp);

template<class For, class T> pair<For, For> equal_range(For first, For last, const T& val);
template<class For, class T, class Cmp>
pair<For, For> equal_range(For first, For last, const T& val, Cmp cmp);
```

Эти алгоритмы соответствуют операциям контейнеров ***multimap*** (§17.4.2). Можно представлять себе алгоритм ***lower_bound*** () как быстрый вариант алгоритмов ***find*** () и ***find_if*** () для сортированных последовательностей. Например:

```
void g(vector<int>& c)
{
  typedef vector<int>::iterator VI;

  VI p = find(c.begin(), c.end(), 7);  // вероятно медленно - O(N); с не нужно сортировать
  VI q = lower_bound(c.begin(), c.end(), 7);  // быстро - O(log(N)); с нужно сортировать
  // ...
}
```

Если вызов ***lower_bound*** (***first, last, k***) не находит ***k***, то он возвращает итератор на первый элемент с ключом, большим ***k***, или ***last***, если такого элемента нет. Такой же способ индикации неудачи используется в ***upper_bound*** () и ***equal_range*** (). А это значит, что мы можем применять эти алгоритмы для того, чтобы определить место вставки элемента, не нарушающее упорядоченного характера последовательности.

18.7.3. Слияние (алгоритмы семейства merge)

При наличии двух отсортированных последовательностей мы можем объединить (слить) их в одну отсортированную последовательность при помощи алгоритма ***merge*** (). А при помощи ***inline_merge*** () можно соединить две части одной последовательности:

```
template<class In, class In2, class Out>
Out merge(In first, In last, In2 first2, In2 last2, Out res);
template<class In, class In2, class Out, class Cmp>
Out merge(In first, In last, In2 first2, In2 last2, Out res, Cmp cmp);

template<class Bi> void inplace_merge(Bi first, Bi middle, Bi last);
template<class Bi, class Cmp> void inplace_merge(Bi first, Bi middlee, Bi last, Cmp cmp);
```

При равенстве элементы первой последовательности будут предшествовать элементам второй последовательности.

Вызов *inplace_merge*() соединяет сортированные последовательности [*f*, *m*) и [*m*, *e*), помещая результат (назад) в [*f*, *e*). Алгоритм *inplace_merge*() полезен в первую очередь тогда, когда у вас есть последовательность, которую можно отсортировать по нескольким критериям. Например, пусть имеется вектор (типа *vector*) с экземплярами рыб, отсортированный по видам рыб (*cod*, *hadddock*, *herring* — треска, морской окунь, сельдь). Если экземпляры рыб одного вида отсортированы по весу, то вы можете отсортировать весь вектор по весу, применив *inplace_merge*() для объединения информации по разным видам рыб (§18.13[20]).

Заметьте, что эти алгоритмы слияния отличаются от методов *merge*() списка *list* (§17.2.2.1) тем, что не удаляют элементы из входных последовательностей. Вместо этого элементы копируются.

18.7.4. Разбиение элементов на две группы (алгоритмы семейства partition)

Разбиение последовательности (partition a sequence) на две группы элементов выполняется так, что все элементы, удовлетворяющие предикату, располагаются перед элементами, предикату не удовлетворяющими. Стандартная библиотека предоставляет алгоритм *stable_partition*(), выполняющий такое разбиение по предикату с сохранением относительного порядка следования элементов в группах (удовлетворяющих предикату и не удовлетворяющих ему). Алгоритм *partition*() делает то же самое, но без сохранения относительного порядка, и он выполняется чуть быстрее в условиях дефицита памяти:

```
template<class Bi, class Pred> Bi partition(Bi first, Bi last, Pred p);
template<class Bi, class Pred> Bi sable_partition(Bi first, Bi last, Pred p);
```

Можно представлять себе алгоритмы разбиения как разновидность сортировки с очень простым критерием. Например:

```
void f(list<Club>& lc)
{
  list<Club>::iterator p = partition(lc.begin(),lc.end(),located_in("København"));
  //...
}
```

Здесь список клубов сортируется так, что клубы из Копенгагена идут первыми. Возвращаемое значение (здесь это *p*) указывает либо на первый элемент, не удовлетворяющий предикату, либо на конец последовательности.

18.7.5. Операции над множествами

Последовательность можно рассматривать как множество. Поэтому имеет смысл обеспечить для последовательностей операции, характерные для множеств, например, объединение и пересечение. Однако эти операции страшно неэффективны, если только последовательности не являются отсортированными. Поэтому стандартная библиотека предоставляет операции над множествами только для отсортированных последовательностей. В частности, операции над множествами пре-

красно работают с контейнерами ***set*** (§17.4.3) и ***multiset*** (§17.4.4), которые сортируются автоматически.

Если же эти операции (алгоритмы) применить к несортированным последовательностям, то результирующие последовательности не будут соответствовать правилам теории множеств. Эти алгоритмы не изменяют свои входные последовательности, а выходные последовательности всегда отсортированы.

Алгоритм ***includes***() проверяет, является ли каждый элемент второй последовательности, то есть последовательности [***first2***, ***last2***[, также и элементом первой последовательности [***first***, ***last***[:

```
template<class In, class In2>
bool includes(In first, In last, In2 first2, In2 last2);

template<class In, class In2, class Cmp>
bool includes(In first, In last, In2 first2, In2 last2, Cmp cmp);
```

Алгоритмы ***set_union***()[1] и ***set_intersection***() порождают свои очевидные результаты в виде отсортированных последовательностей:

```
template<class In, class In2, class Out>
Out set_union(In first, In last, In2 first2, In2 last2, Out res);
template<class In, class In2, class Out, class Cmp>
Out set_union(In first, In last, In2 first2, In2 last2, Out res, Cmp cmp);

template<class In, class In2, class Out>
Out set_intersection(In first, In last, In2 first2, In2 last2, Out res);
template<class In, class In2, class Out, class Cmp>
Out set_intersection(In first, In last, In2 first2, In2 last2, Out res, Cmp cmp);
```

Алгоритм ***set_difference***() порождает последовательность, чьи элементы являются членами первой, но не второй входной последовательности. Алгоритм ***set_symmetric_difference***() порождает последовательность элементов, входящих в одну и только одну из входных последовательностей:

```
template<class In, class In2, class Out>
Out set_difference(In first, In last, In2 first2, In2 last2, Out res);
template<class In, class In2, class Out, class Cmp>
Out set_difference(In first, In last, In2 first2, In2 last2, Out res, Cmp cmp);

template<class In, class In2, class Out>
Out set_symmetric_difference(In first, In last, In2 first2, In2 last2, Out res);
template<class In, class In2, class Out, class Cmp>
Out set_symmetric_difference(In first, In last, In2 first2, In2 last2, Out res, Cmp cmp);
```

Например:

```
char v1[] = "abcd";
char v2[] = "cdef";

void f(char v3[])
{
  set_difference(v1,v1+4,v2,v2+4,v3);              // v3 = "ab"
  set_symmetric_difference(v1,v1+4,v2,v2+4,v3);    // v3 = "abef"
}
```

[1] Union — объединение, intersection — пересечение. — *Прим. ред.*

18.8. «Кучи»

Слово «куча» (*heap*) означает разное в разных контекстах. Применительно к алгоритмам слово «куча» часто означает такую организацию последовательности[1], когда ее первый элемент является и максимальным элементом, а добавление и удаление элементов (используются функции ***push_heap***() и ***pop_heap***()) достаточно быстры и в худшем случае оцениваются как ***O***(***log***(***N***)), где ***N*** — число элемнтов последовательности. Затраты на сортировку (функцией ***sort_heap***()) в худшем случае оценивается как ***O***(***N****log*(***N***)). Кучи реализуются следующим набором функций:

```
template<class Ran> void push_heap(Ran first, Ran last);
template<class Ran, class Cmp> void push_heap(Ran first, Ran last, Cmp cmp);

template<class Ran> void pop_heap(Ran first, Ran last);
template<class Ran, class Cmp> void pop_heap(Ran first, Ran last, Cmp cmp);

template<class Ran> void make_heap(Ran first, Ran last); // последовательность - в кучу
template<class Ran, class Cmp> void make_heap(Ran first, Ran last, Cmp cmp);

template<class Ran> void sort_heap(Ran first, Ran last); // кучу - в последовательность
template<class Ran, class Cmp> void sort_heap(Ran first, Ran last, Cmp cmp);
```

Стиль этих алгоритмов довольно странен. Лучше было бы предоставить адаптерный класс с четырьмя операциями; получилось бы что-нибудь вроде ***priority_queue*** (§17.3.3). На самом деле, ***priority_queue*** почти всегда и реализуется с помощью кучи.

Элемент, добавляемый при вызове ***push_heap***(***first, last***) есть *(***last-1***). Предполагается, что [***first, last-1***) уже является кучей, так что ***push_heap***() расширяет кучу до [***first, last***) добавлением следующего элемента. Таким образом, можно построить кучу из существующей последовательности путем последовательных операций ***push_heap***(). И наоборот, ***pop_heap***(***first, last***) удаляет первый элемент из кучи, переставляя его с последним элементом *(***last-1***) и делая последовательность [***first, last-1***) кучей.

18.9. Нахождение минимума и максимума

Описанные здесь алгоритмы выбирают значение на основе сравнения. Ясно, что нахождение максимума и минимума из двух значений имеет очевидную практическую пользу:

```
template<class T> const T& max(const T& a, const T& b)
{
  return (a<b) ? b : a;
}

template<class T, class Cmp> const T& max(const T& a, const T& b, Cmp cmp)
{
  return (cmp(a,b)) ? b : a;
}

template<class T > const T& min(const T& a, const T& b);
template<class T, class Cmp> const T& min(const T& a, const T& b, Cmp cmp);
```

[1] Часто это бинарное дерево, реализованное в виде некоторой последовательной коллекции. — *Прим. ред.*

Операции ***min***() и ***max***() имеют очевидные обобщения для работы с последовательностями:

```
template<class For> For max_element(For first, For last);
template<class For, class Cmp> For max_element(For first, For last, Cmp cmp);

template<class For> For min_element(For first, For last);
template<class For, class Cmp> For min_element(For first, For last, Cmp cmp);
```

Наконец, лексикографическое упорядочение строк символов обобщается на случай последовательности значений типа, располагающего необходимым сравнением:

```
template<class In, class In2>
bool lexicographical_compare(In first, In last, In2 first2, In2 last2);

template<class In, class In2, class Cmp>
bool lexicographical_compare(In first, In last, In2 first2, In2 last2, Cmp cmp)
{
  while(first!=last && first2!=last2)
  {
    if(cmp(*first,*first2)) return true;
    if(cmp(*first2++,*first++)) return false;
  }
  return first==last && first2!=last2;
}
```

Это сильно напоминает функцию сравнения строк в §13.4.1. Однако алгоритм ***lexicographical_compare***() сравнивает последовательности вообще, а не только лишь строки. Он также возвращает ***bool***, а не более распространенный (и в общем случае более полезный) ***int***. Возврат равен ***true***, только если первая последовательность меньше второй в смысле операции <. При равенстве последовательностей возвращается ***false***.

С-строки и строки типа ***string*** являются последовательностями, так что ***lexicographical_compare***() может использоваться для их сравнения. Например:

```
char v1[] = "yes";
char v2[] = "no";
string s1 = "Yes";
string s2 = "No";

void f()
{
  bool b1 = lexicographical_compare(v1,v1+strlen(v1),v2,v2+strlen(v2));
  bool b2 = lexicographical_compare(s1.begin(),s1.end(),s2.begin(),s2.end());

  bool b3 = lexicographical_compare(v1,v1+strlen(v1),s1.begin(),s1.end());
  bool b4 =
lexicographical_compare(s1.begin(),s1.end(),v1,v1+strlen(v1),Nocase());
}
```

Последовательности не обязаны быть одного и того же типа. Требуется лишь возможность сравнения их элементов и, конечно, сам критерий сравнения. Это делает ***lexicographical_compare***() более универсальным и потенциально более медленным, чем сравнение для типа string. См. также §20.3.8. Nocase определяется в §17.1.4.1.

18.10. Перестановки (permutations)

Последовательность из четырех элементов можно переупорядочить (задать порядок следования элементов) 4*3*2*1 способами. Каждый из получающихся при этом вариантов называется *перестановкой* (*permutation*). Например, для четырех символов ***abcd*** можно получить 24 перестановки:

abcd abdc acbd acdb adbc adcb bacd badc
bcad bcda bdac bdca cabd cadb cbad cbda
cdab cdba dabc dacb dbac dbca dcab dcba

Функции ***next_permutation*** () и ***prev_permutation*** () порождают такие перестановки последовательности:

```
template<class Bi> bool next_permutation (Bi first, Bi last) ;
template<class Bi, class Cmp> bool next_permutation (Bi first, Bi last, Cmp cmp) ;

template<class Bi> bool prev_permutation (Bi first, Bi last) ;
template<class Bi, class Cmp> bool prev_permutation (Bi first, Bi last, Cmp cmp) ;
```

Все перестановки строки ***abcd*** можно произвести следующим образом:

```
int main ()
{
  char v[] = "abcd";
  cout<< v << '\t';
  while (next_permutation (v,v+4) ) cout<< v << '\t';
}
```

Перестановки производятся в лексикографическом порядке (§18.9). Возврат функции ***next_permutation*** () показывает, существует ли следующая перестановка. Возврат ***false*** означает, что не существует следующей перестановки и что последовательность находится в лексикографически упорядоченном состоянии. Возврат функции ***prev_permutation*** () показывает, существует ли предыдущая перестановка. Возврат ***false*** означает, что не существует предыдущей перестановки и что последовательность находится в обратном лексикографически упорядоченном состоянии.

18.11. Алгоритмы в С-стиле

От стандартной библиотеки языка С стандартная библиотека С++ унаследовала несколько алгоритмов, имеющих дело с С-строками (§20.4.1), а также быструю сортировку и бинарный поиск, применяемую исключительно для массивов.

Функции ***qsort*** () и ***bsearch*** () представлены в заголовочных файлах <***cstdlib***> и <***stdlib.h***>. Они работают с массивами из n элементов размера ***elem_size***, используя операцию сравнения «меньше чем», передаваемую как указатель на функцию. Элементы должны быть типа, не имеющего определенных пользователем копирующего конструктора, деструктора и операции присваивания:

```
typedef int (*__cmp) (const void*, const void*) ;

void qsort (void* p, size_t n, size_t elem_size, __cmp) ;                  // сортирует p
void* bsearch (const void* key, void* p, size_t n, size_t elem_size, __cmp) ; // ищет key в p
```

Применение функции *qsort*() продемонстрировано в §7.7.

Эти алгоритмы предоставляются исключительно с целью обеспечения совместимости с языком С; *sort*() (§18.7.1) и *search*() (§18.5.5) более универсальны и почти всегда более эффективны.

18.12. Советы

1. Предпочитайте алгоритмы циклам; §18.5.1.
2. Когда пишите цикл, подумайте, нельзя ли заменить его универсальным алгоритмом; §18.2.
3. Регулярно пересматривайте весь набор алгоритмов в поисках новых областей их применения; §18.2.
4. Всегда обращайте внимание на то, чтобы пара аргументов-итераторов относилась к одной и той же последовательности (определяла последовательность); §18.3.1.
5. Разрабатывайте программы так, чтобы наиболее употребительные операции были простыми и безопасными; §18.3, §18.3.1.
6. Выражайте условия таким образом, чтобы их можно было использовать как предикаты; §18.4.2.
7. Помните, что предикаты есть функции или объекты, но не типы; §18.4.2.
8. Вы можете применить «связыватели», чтобы получить унарные предикаты из бинарных предикатов; §18.4.4.1.
9. Если для алгоритмов нужно использовать функции-члены классов элементов контейнера, используйте *mem_fun*() и *mem_fun_ref*(); §18.4.4.2.
10. Для связывания аргумента-функции применяйте *ptr_fun*(); §18.4.4.3.
11. Помните, что *strcmp*() отличается от операции == тем, что при равенстве возвращает *0*; §18.4.4.4.
12. Используйте *for_each*() и *transform*() только тогда, когда отсутствуют более специфические для задачи алгоритмы; §18.5.1.
13. Для алгоритмов с разными критериями сравнения и равенства используйте предикаты; §18.4.2.1, §18.6.3.1.
14. Используйте предикаты и другие функциональные объекты для расширения областей применения алгоритмов; §18.4.2.
15. Умолчательные операции < и == над указателями редко подходят для стандартных алгоритмов; §18.6.3.1.
16. Алгоритмы напрямую не добавляют и не удаляют элементы из своих входных аргументов-последовательностей; §18.6.
17. Убедитесь, что используемые для последовательности операции «меньше чем» и «равно» соответствуют друг другу; §18.6.3.1.
18. Для повышения элегантности и эффективности можно применить сортированные последовательности; §18.7.
19. Используйте *qsort*() и *bsearch*() только с целью совместимости; §18.11.

18.13. Упражнения

Решение некоторых задач можно найти, посмотрев реализацию стандартной библиотеки. Сделайте себе полезное одолжение: не пытайтесь смотреть сторонний код до того, как вы сами попробовали решить задачу.

1. (*2) Изучите *O*() нотацию. Приведите реалистичный пример, в котором получается, что *O*(*N***N*) быстрее, чем *O*(*N*) для некоторых *N*>*10*.
2. (*2) Реализуйте и протестируйте четыре функции ***mem_fun***() и ***mem_fun_ref***() (§18.4.4.2).
3. (*1) Напишите алгоритм ***match***(), подобный алгоритму ***mismatch***(), за исключением того, что возвращает итераторы на первые два соответствующих элемента, удовлетворяющих предикату.
4. (*1.5) Реализуйте и протестируйте функцию ***Print_name*** из §18.5.1.
5. (*1) Отсортируйте ***list*** при помощи стандартных библиотечных алгоритмов.
6. (*2.5) Определите версии ***iseq***() (§18.3.1) для встроенных массивов, ***istream*** и пар итераторов. Задайте подходящий набор перегруженных немодифицирующих последовательность алгоритмов (§18.5) для работы с ***Iseq***. Обсудите, как избежать неоднозначностей и сильного роста числа функциональных шаблонов.
7. (*2) Дополнительно к ***iseq***() определите ***oseq***. Выходная последовательность, которая задается как аргумент ***oseq***(), должна замещаться выходной последовательностью использующего ***oseq***() алгоритма. Определите подходящий набор перегрузок для по крайней мере трех стандартных алгоритмов по вашему выбору.
8. (*1.5) Создайте вектор (типа ***vector***) квадратов целых чисел от 1 до 100. Распечатайте таблицу квадратов. Вычисляйте корень квадратный из элементов *вектора* и распечатайте новый вектор.
9. (*2) Напишите набор функциональных объектов, выполняющих побитовые логические операции над своими операндами. Проверьте эти объекты на векторах с элементами типа ***char***, ***int*** и ***bitset<67>***.
10. (*1) Напишите связывающий адаптер ***binder3***(), который должен связывать второй и третий аргументы трехаргументной функции для получения унарного предиката. Приведите пример полезного применения ***binder3***().
11. (*1.5) Напишите маленькую программу, которая удаляет одинаковые соседние слова в файле.
12. (*2.5) Определите формат записи, содержащей ссылки на статьи и книги в файле. Напишите программу, которая могла бы читать из файла записи по времени издания, имени автора, ключевому слову в названии или по издательству. Пользователь должен иметь возможность потребовать выдачи результатов в отсортированном по аналогичным критериям порядке.
13. (*2) Реализуйте алгоритм ***move***() в стиле ***copy***() таким образом, чтобы допускалось перекрытие входной и выходной последовательностей. Обеспечьте приемлемую эффективность в случае аргументов-итераторов произвольного доступа.

14. (*1.5) Создайте все анаграммы для слова ***food***. То есть получите все четырехбуквенные комбинации из букв ***f***, ***o***, ***o***, ***d***. Обобщите программу так, чтобы она на входе получала слово, а на выходе выдавала анаграмму.
15. (*1.5) Напишите программу, которая бы выдавала анаграммы предложений, то есть все перестановки слов предложения (а не перестановки букв в словах).
16. (*1.5) Реализуйте ***find_if***() (§18.5.2), а затем с его помощью и ***find***(). Найдите способ реализации этих функций в варианте с одинаковыми именами.
17. (*2) Реализуйте ***search***() (§18.5.5). Обеспечьте оптимизированную версию для итераторов произвольного доступа.
18. (*2) Возьмите алгоритм сортировки из стандартной библиотеки (например, ***sort***()) или вариант с сортировкой Шелла (§13.5.2) и вставьте в них код, распечатывающий сортируемую последовательность после каждого акта обмена элементов местами.
19. (*2) Алгоритм ***sort***() не работает с двунаправленными итераторами. Существует гипотеза, что копирование в вектор и последующая его сортировка выполняются быстрее, чем сортировка последовательности с двунаправленными итераторами. Реализуйте сортировку для двунаправленных итераторов и проверьте гипотезу.
20. (*2.5) Представьте, что вы ведете записи о группе спортсменов-рыболовов. Для каждого улова записывайте вид рыбы, ее вес, дату, фамилию рыболова и т.д. Отсортируйте и распечатайте записи в соответствии с разными критериями. Подсказка: ***inplace_merge***().
21. (*2) Создайте список студентов, изучающих математику, английский, французский и биологию. Выберите по 20 фамилий из 40 для каждого предмета. Перечислите студентов, изучающих как математику, так и английский. Составьте список студентов, изучающих французский, но не биологию или математику. Выявите студентов, не изучающих точные науки. Перечислите студентов, изучающих математику и французский, но не изучающих ни английский, ни биологию.
22. (*1.5) Напишите функцию ***remove***(), которая на самом деле удаляла бы элементы из контейнера.

19

Итераторы и аллокаторы

Причина, по которой структуры данных и алгоритмы состыковываются бесшовным образом, состоит в том, что они ... ничего не знают друг о друге.
— Алекс Степанов

Итераторы и последовательности — операции над итераторами — шаблон ***iterator_traits*** — категории итераторов — итераторы для вставок — обратные итераторы — итераторы потоков — итераторы с проверкой — исключения и алгоритмы — аллокаторы — стандартный *аллокатор* — пользовательский аллокатор — низкоуровневые функции для работы с памятью — советы — упражнения.

19.1. Введение

Итераторы — это тот клей, который соединяет воедино контейнеры и алгоритмы. Они предоставляют столь абстрактный вид на данные, что разработчикам алгоритмов не приходится заботиться о конкретных деталях устройства многочисленных структур данных. И наоборот, стандартная модель доступа к данным, предоставляемая итераторами, освобождает контейнеры от необходимости обеспечивать дополнительные наборы операций доступа. Аналогичным образом, аллокаторы (распределители памяти) изолируют реализации контейнеров от деталей работы с памятью.

Итераторы поддерживают абстрактную модель данных в виде последовательности объектов (§19.2). Аллокаторы же обеспечивают отображение низкоуровневой модели данных как последовательности байтов в высокоуровневую объектную модель (§19.4). В свою очередь, стандартная низкоуровневая модель памяти поддерживается небольшим количеством стандартных функций (§19.4.4).

Итераторы — это понятие, с которым должен быть знаком каждый программист. И напротив, аллокаторы представляют собой механизм, о котором программисты могут в общем случае не беспокоиться, так как на практике мало кому придется писать свой собственный аллокатор.

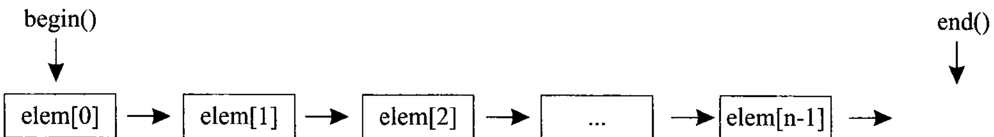
19.2. Итераторы и последовательности

Итератор — это чистая абстракция. Так что все, что работает как итератор, и есть итератор (§3.8.2). Итератор служит абстракцией для указателя на элемент последовательности. Вот ключевые концепции этого понятия:

- «текущий адресуемый элемент» (разыменование, определяемое операциями * и ->)
- «указать на следующий элемент» (определяется операцией инкремента ++)
- «равенство» (определяемое операцией ==)

Например, встроенный тип ***int**** есть итератор для ***int***[], а класс ***list<int>::iterator*** есть итератор для списка ***list<int>***.

Последовательность служит абстракцией для «чего-либо, где можно двигаться от начала к концу с помощью операции получения следующего элемента»:



Примерами последовательностей служат массивы (§5.2), векторы (§16.3), односвязные списки (§17.8[17]), двусвязные списки (§17.2.2), деревья (§17.4.1), потоки ввода (§21.3.1) и потоки вывода (§21.2.1). Для каждого случая имеются свои собственные итераторы.

Итераторные классы и функции объявляются в пространстве имен ***std*** и расположены в заголовочном файле ***<iterator>***.

Итератор — это в общем случае не указатель. Это, скорее, абстракция указателя на массив. Понятия «нулевого итератора» не существует. Для того, что узнать, указывает ли итератор на некоторый элемент или нет, его стандартным образом сравнивают с *концом данной последовательности* (*end of sequence*), а не путем выявления *нулевого элемента* (*null element*). Такая процедура упрощает алгоритмы, избавляя от необходимости особой обработки конца последовательности, и она хорошо обобщается на случай последовательностей произвольных типов.

Про итератор, указывающий на реальный элемент последовательности, говорят как о *действительном итераторе* (*valid iterator*), и его можно разыменовать (используя *, [] или ->, соответственно). Недействительным итератор может быть из-за того, что он не был инициализирован; из-за того, что он указывает на контейнер, размеры которого были явно или неявно изменены (§16.3.6, §16.3.8); из-за уничтожения контейнера или потому что он обозначает конец последовательности (§18.2). Конец последовательности можно представлять себе как итератор, указывающий на гипотетический *элемент*, расположенный в памяти компьютера *за последним элементом последовательности* (*past-the-last element*).

19.2.1. Операции над итераторами

Разные виды итераторов поддерживают разные наборы операций. Например, чтение требует иных операций, чем запись; итераторы для vector допускают удобные и эффективные операции произвольного доступа, которые были бы слишком

дорогими для ***list*** или ***istream***. Как следствие мы разбиваем итераторы на пять категорий в соответствии с наборами операций, которые они могут поддерживать эффективно (за постоянное время; §17.1):

Итераторы: операции и категории					
Категория:	output (вывода)	input (ввода)	forward (прямые)	bidirectional (двунаправленные)	random-access (произвольного доступа)
Сокращение:	***Out***	***In***	***For***	***Bi***	***Ran***
Чтение:		=*p	=*p	=*p	=*p
Доступ:		->	->	->	-> []
Запись:	*p=		*p=	*p=	*p=
Итерация:	++	++	++	++ —	++ — + - += -=
Сравнение:		== !=	== !=	== !=	== ! < > >= <=

Как чтение, так и запись выполняются через разыменовывание итератора операцией *:

```
*p = x;      // пишем значение x через p
x = *p;      // читаем в переменную x через p
```

Чтобы быть итераторным типом, любой тип должен реализовывать соответствующий итераторам набор операций, причем операции должны иметь заданный предопределенный смысл. Это означает, что каждая операция должна порождать тот же самый эффект, что и для обычного указателя.

Независимо от категории итераторы предоставляют как *константный*, так и *неконстантный* виды доступа к указуемым ими элементам. Для итераторов *константного* доступа не допускается запись элементов. Но хоть итераторы и обеспечивают необходимый набор операций, окончательное решение о допустимости тех или иных действий над элементом остается за самим типом элементов.

Операции чтения и записи приводят к копированию объектов. Поэтому типы элементов должны реализовывать традиционную семантику копирования (§17.1.4).

Только итераторы произвольного доступа допускают сложение (вычитание) с целыми числами для реализации относительной адресации. Тем не менее, за исключением итераторов вывода всегда можно определить расстояние между двумя итераторами в процессе итерирования элементов последовательности, для чего и предназначена функция ***distance***():

```
template<class In> typename iterator_traits<In>::difference_type
distance(In first, In last)
{
  typename iterator_traits<In>::difference_type d = 0;
  while(first++ != last) d++;
  return d;
}
```

Тип ***iterator_traits<In>::difference_type*** для любого итератора ***In*** определяет расстояние между элементами (§19.2.2).

Функция названа ***distance***(), а не ***operator-***(), поскольку она может оказаться довольно дорогой, в то время как все операции над итераторами выполняются за постоянное время (§17.1).

Последовательный обсчет всех элементов — это не то, что можно нечаянно позволить себе для большой последовательности. Для итераторов произвольного доступа в стандартной библиотеке имеется существенно более эффективная реализация функции ***distance***().

Аналогично, функция ***advance***() потенциально медленнее операции +=:

```
template<class In, class Dist> void advance(In& i, Dist n);      // i += n
```

19.2.2. Шаблон iterator_traits

Мы используем итераторы для получения информации об указуемых объектах и о последовательностях, на которых они действуют. Например, мы можем разыменовать итератор и манипулировать целевым объектом, а также можем определить число элементов последовательности, отталкиваясь от работающих с ней итераторов. Для работы с такими операциями нам нужно ссылаться на типы, связанные с итераторами, как на «тип объекта, указуемого итератором» и «тип расстояния между двумя итераторами». Связанные с итераторами типы описываются небольшим набором объявлений в рамках шаблонного класса ***iterator_traits***:

```
template<class Iter> struct iterator_traits
{
  typedef typename Iter::iterator_category iterator_category;   // §19.2.3
  typedef typename Iter::value_type value_type;                 // тип элемента
  typedef typename Iter::difference_type difference_type;
  typedef typename Iter::pointer pointer;          // тип возврата для operator->()
  typedef typename Iter::reference reference;      // тип возврата для operator*()
};
```

Тип ***difference_type*** — это тип разности двух итераторов, а тип ***iterator_category*** показывает, какие операции поддерживает итератор. Для обычных указателей предоставляются специализации (§13.5) для <***T****> и <***const T****>. В частности:

```
template<class T> struct iterator_traits<T*>      // специализация для указателей
{
  typedef random_access_iterator_tag iterator_category;
  typedef T vale_type;
  typedef ptrdiff_t difference_type;
  typedef T* pointer;
  typedef T& reference;
};
```

Таким образом, разность между двумя указателями представляется стандартным библиотечным типом ***ptrdiff_t*** из заголовочного файла <***cstddef***> (§6.2.1), а ***pointer*** обеспечивает произвольный доступ (§19.2.3). Располагая шаблоном ***iterator_traits*** мы можем написать код, зависящий от свойств итератора-параметра. Алгоритм ***count***() служит классическим примером:

```
template<class In, class T>
typename iterator_traits<In>::difference_type count(In first, In last, const T& val)
```

```
{
  typename iterator_traits<In>::difference_type res = 0;
  while (first != last) if (*first++ == val) ++res;
  return res;
}
```

Тип возврата определяется через ***iterator_traits***<***In***>. Причина состоит в том, что нет никаких языковых примитивов, позволяющих напрямую выразить произвольный тип в виде комбинации других типов, например, «тип результата вычитания двух операндов типа ***In***».

Вместо применения ***iterator_traits*** мы могли бы применить специализации ***count***() для указателей:

```
template<class In, class T> typename In::difference_type
count(In first, In last, const T& val);

template<class In, class T> ptrdiff_t count<T*, T> (T* first, T* last, const T& val);
```

Таким образом, однако, проблема решается лишь для ***count***(). Примени мы этот прием для дюжины алгоритмов, информация о типе разности итераторов реплицировалась бы столько же раз, а как правило, любое проектное решение лучше сосредоточить в одном месте (§23.4.2). Тогда и исправлять что-либо в нем пришлось бы также в одном месте.

Поскольку ***iterator_traits***<***Iterator***> определен для всех итераторов, мы неявно определяем ***iterator_traits*** всякий раз, когда создаем новый итераторный тип. Если для нового итераторного типа не подходит то, что автоматически порождается от общего шаблона ***iterator_traits***, мы обеспечиваем специализацию в том же ключе, что и указанная выше стандартная специализация для указателей. При автоматическом порождении от ***iterator_traits*** предполагается, что итератор является классом с типами-членами ***difference_type***, ***value_type*** и т.д. Стандартная библиотека в заголовочном файле <***iterator***> определяет базовый тип, который можно использовать для определения указанных типов:

```
template<class Cat, class T, class Dist = ptrdiff_t, class Ptr = T*, class Ref = T&>
struct iterator
{
  typedef Cat iterator_category;    // §19.2.3
  typedef T value_type;             // тип элемента
  typedef Dist difference_type;     // тип разности итераторов
  typedef Ptr pointer;              // тип возврата для ->
  typedef Ref reference;            // тип возврата для *
};
```

Заметьте, что ***reference*** и ***pointer*** — это не итераторы. Они предназначены для возвратов ***operator****() и ***operator***->() некоторых итераторов.

Шаблон ***iterator_traits*** имеет ключевую важность для упрощения некоторых интерфейсов, зависящих от итераторов, и для эффективной реализации многих алгоритмов.

19.2.3. Категории итераторов

Различные виды итераторов — называемые категориями итераторов — укладываются в следующую иерархию:

Это вовсе не диаграмма наследования классов. Категории итераторов просто классифицируют их по наборам поддерживаемых операций. Множество не связанных друг с другом типов могут попадать в одну и ту же категорию итераторов. Например, и обычные указатели (§19.2.2), и *Checked_iters* (§19.3) являются итераторами произвольного доступа (random access).

Как отмечено в главе 18, разные алгоритмы принимают в качестве параметров итераторы разного вида, а один и тот же алгоритм в принципе можно реализовать с разной эффективностью, применяя разные виды итераторов. Для поддержки разрешения перегрузки для разных видов итераторов стандартная библиотека предоставляет пять классов, соответствующих пяти итераторным категориям:

```
struct input_iterator_tag { } ;
struct output_iterator_tag { } ;
struct forward_iterator_tag : public input_iterator_tag { } ;
struct bidirectional_iterator_tag : public forward_iterator_tag { } ;
struct random_access_iterator_tag : public bidirectional_iterator_tag { } ;
```

Глядя на операции, поддерживаемые итераторами ввода и прямыми итераторами (§19.2.1), можно предположить, что *forward_iterator_tag* наследует как от *output_iterator_tag*, так и от *input_iterator_tag*. Причины того, что это не так, запутанны и не совсем логичны. Однако я еще не видел примера, где такое наследование упростило бы написание реального кода.

Наследование теговых классов (tag-classes) полезно лишь в случае необходимости избавиться от множества вариантов функций, в которых разные виды итераторов (но не все) используются с одними и теми же алгоритмами. Рассмотрим, как реализовать *distance*():

```
template<class In> typename iterator_traits<In>::difference_type
distance (In first, In last) ;
```

Есть две очевидные альтернативы:

1. Если *In* есть итератор произвольного доступа, вычесть *first* из *last*.
2. Иначе инкрементировать итератор от *first* до *last* и вычислить расстояние.

Эти альтернативы можно реализовать парой вспомогательных функций:

```
template<class In>
typename iterator_traits<In>::difference_type
dist_helper (In first, In last, input_iterator_tag)
{
  typename iterator_traits<In>::difference_type d = 0;
```

```
  while (first++ != last) d++;      // используем только инкремент
  return d;
}

template<class Ran>
typename iterator_traits<Ran>::difference_type
dist_helper(Ran first, Ran last, random_access_iterator_tag)
{
  return last-first;                // полагаемся на произвольный доступ
}
```

Указанные с помощью теговых классов типы итераторов-аргументов позволяют явным образом заявить, какой вид итераторов ожидается. Категории итераторов используются исключительно при разрешении перегрузки; они не используются в реальных вычислениях. Так что этот механизм имеет отношение исключительно к этапу компиляции. Дополнительно к автоматическому выбору вспомогательной функции данная технология позволяет произвести немедленную проверку типов (§13.2.5).

Теперь нам будет просто определить ***distance***() посредством вызова соответствующей вспомогательной функции:

```
template<class In> typename iterator_traits<In>::difference_type
distance(In first, In last)
{
  return dist_helper(first, last, iterater_traits<In>::iterator_category());
}
```

Чтобы вызвать ***dist_helper***(), ***iterator_traits<In>::iterator_category*** должно быть либо ***input_iterator_tag***, либо ***random_access_iterator_tag***. Однако нет необходимости иметь разные версии ***dist_helper***() для прямых и двунаправленных итераторов. Благодаря наследованию теговых классов эти случаи успешно обрабатываются ***dist_helper***(), принимающей в качестве аргумента ***input_iterator_tag***. Отсутствие версии для ***output_iterator_tag*** отражает факт бессмысленности ***distance***() для итераторов вывода:

```
void f( vector<int>& vi, list<double>& ld,
        istream_iterator<string>& is1, istream_iterator<string>& is2,
        ostream_iterator<char>& os1, ostream_iterator<char>& os2 )
{
  distance(vi.begin(), vi.end());   // алгоритм с вычитанием
  distance(ld.begin(), ld.end());   // алгоритм с инкрементом
  distance(is1, is2);               // алгоритм с инкрементом
  distance(os1, os2);               // error: ошибка в категории итератора,
                                    // аргументы неверного типа для dist_helper()
}
```

В реальной программе, впрочем, вызов ***distance***() для ***istream_iterator*** вряд ли имеет большое значение. Эффектом будет чтение входа с его последующим отбрасыванием и возврат числа отброшенных значений.

Применение ***iterator_traits<T>::iterator_category*** позволяет программисту предоставлять альтернативные реализации таким образом, что для пользователя (которому нет дела до деталей реализации) выбор подходящей для его структуры данных

альтернативы выполняется автоматически. Другими словами, это помогает элегантно спрятать детали реализации за публичным интерфейсом. Встраивание же (inlining) позволяет получить эту элегантность без дополнительных накладных расходов во время выполнения программы.

19.2.4. Итераторы для вставок

Попытка направить выходной поток данных внутрь контейнера в место, указуемое итератором, предполагает, что последующие элементы контейнера могут быть переписаны. Это также может приводить и к переполнению, и к порче памяти. Например:

```
void f(vector<int>& vi)
{
  fill_n(vi.begin(), 200, 7);        // присваиваем 7 всем элементам vi[0]..[199]
}
```

Если в контейнере ***vi*** элементов меньше ***200***, то возникают проблемы.

Для решения подобного рода проблем стандартная библиотека определяет в заголовочном файле <***iterator***> три шаблонных класса итераторов вставки[1] и три вспомогательных функции, упрощающих их использование:

```
template<class Cont> back_insert_iterator<Cont> back_inserter(Cont& c);
template<class Cont> front_insert_iterator<Cont> front_inserter(Cont& c);
template<class Cont, class Out> insert_iterator<Cont> inserter(Cont& c, Out p);
```

При помощи функции ***back_inserter***() элементы вставляются (добавляются) в конец контейнера; при помощи функции ***front_inserter***() — в начало контейнера, а при помощи «просто» ***inserter***() — в место, указуемое ее аргументом-итератором. Для вызова ***inserter***(***c***, ***p***) ***p*** должно быть действительным для с итератором. Естественно, что с каждой вставкой контейнер растет.

Итераторы вставок осуществляют вставку новых элементов в последовательность, применяя ***push_back***(), ***push_front***() или ***insert***() (§16.3.6), а не посредством переписывания существующих в контейнере значений. Например:

```
void g(vector<int>& vi)
{
  fill_n(back_inserter(vi), 200, 7);  // добавляем 200 раз по 7 в конец vi
}
```

Итераторы вставок столь же просты и эффективны, сколь и полезны. Например:

```
template<class Cont>
class insert_iterator: public iterator<output_iterator_tag, void, void, void, void>
{
protected:
  Cont& container;                    // контейнер для вставки
  typename Cont::iterator iter;       // указывает на контейнер

public:
  explicit insert_iterator(Cont& x, typename Cont::iterator i)
        : container(x), iter(i) {}
```

[1] По стандартной классификации относятся к итераторным адаптерам. — *Прим. ред.*

```
    insert_iterator& operator= (const typename Cont::value_type& val)
    {
      iter = container.insert (iter, val) ;
      ++iter;
      return *this;
    }

    insert_iterator& operator* () { return *this; }
    insert_iterator& operator++ () { return *this; }        // префиксный ++
    insert_iterator operator++ (int) { return *this; }      // постфиксный ++
};
```

Ясно, что итераторы вставок — это итераторы вывода.

Класс *insert_iterator* — это особый случай выходной последовательности. По аналогии с *iseq* из §18.3.1 мы могли бы определить

```
template<class Cont>
insert_iterator<Cont>
oseq (Cont& c, typename Cont::iterator first, typename Cont::iterator last)
{
  return insert_iterator<Cont> (c, c.erase (first, last)) ; // erase разъясняется в §16.3.6
}
```

Ясно, что выходная последовательность удаляет старые элементы и заменяет их на элементы вывода. Например:

```
void f(list<int>& li, vector<int>& vi)      // заменяет вторую половину vi копией li
{
  copy (li.begin () , li.end () , oseq (vi, vi.begin () +vi.size () /2, vi.end () ) ) ;
}
```

Аргументом *oseq*() должен быть именно контейнер, ибо уменьшить размеры контейнера невозможно, если располагать лишь итераторами на его элементы (§18.6, §18.6.3).

19.2.5. Обратные итераторы

Стандартные контейнеры предоставляют функции-члены *rbegin*() и *rend*() для итерирования элементов в обратном порядке (§16.3.2). Эти функции возвращают обратные итераторы[1] типа *reverse_iterator*:

```
template <class Iter>
class reverse_iterator:
                public iterator<typename iterator_traits<Iter>::iterator_category,
                            typename iterator_traits<Iter>::valuy_type,
                            typename iterator_traits<Iter>::difference_type,
                            typename iterator_traits<Iter>::pointer,
                            typename iterator_traits<Iter>::reference>
{
protected:
  Iter current; //current указывает на элемент, следующий за указуемым через *this

public:
  typedef Iter iterator_type;
reverse_iterator () : current () { }
```

[1] Относятся к итераторным адаптерам. — *Прим. ред.*

```
    explicit reverse_iterator(Iter x) : current(x) { }
    template<class U> reverse_iterator(const reverse_iterator<U>& x) :
  current(x.base()) { }

    Iter base() const {return current; }    // значение текущего итератора

    reference operator*() const {Iter tmp = current; return *--tmp; }
    pointer operator->() const;
    reference operator[](difference_type n) const;

    reverse_iterator& operator++() {--current; return *this; }
    reverse_iterator operator++(int) {reverse_iterator t = current; --current; return t; }
    reverse_iterator& operator--() {++current; return *this; }
    reverse_iterator operator--(int) {reverse_iterator t = current; ++current; return t; }

    reverse_iterator operator+(difference_type n) const;
    reverse_iterator& operator+=(difference_type n);
    reverse_iterator operator-(difference_type n) const;
    reverse_iterator& operator-=(difference_type n);
  };
```

Класс ***reverse_iterator*** реализуется с применением поля ***current*** итераторного типа. Итераторное поле ***current*** может указывать лишь на элементы своей последовательности плюс элемент «за последним элементом последовательности» (one-past-the-end element). Но для класса ***reverse_iterator*** этот элемент является элементом «перед первым элементом исходной последовательности», и к которому доступа нет. Чтобы избежать проблем с доступом, поле ***current*** ссылается на элемент, расположенный за элементом, на который ссылается сам ***reverse_iterator***. Для этого операция * возвращает значение *(***current-1***), а операция ++ реализуется с помощью операции -- над полем ***current***.

Класс ***reverse_iterator*** поддерживает лишь те операции, которые поддерживаются возвращающим объект этого класса контейнером. Например:

```
  void f(vector<int>& v, list<char>& lst)
  {
    v.rbegin()[3] = 7;          // ok: итератор произвольного доступа
    lst.rbegin()[3] = '4';      // error: двунаправленный итератор не поддерживает []
    *(++++++lst.rbegin()) = '4';  // ok!
  }
```

Кроме того, для обратных итераторов стандартная библиотека предоставляет операции ==, !=, <, <=, >, >=, + и -.

19.2.6. Потоковые итераторы

Обычно ввод/вывод выполняется с помощью библиотек потоков (глава 21), графического интерфейса пользователя (не стандартизованного в C++) или с помощью библиотек ввода/вывода языка C (§21.8). Все эти средства ввода/вывода ориентированы в первую очередь на чтение/запись индивидуальных значений различных типов. Стандартная библиотека предоставляет четыре итераторных типа[1] для согласования потокового ввода/вывода с контейнерами и алгоритмами:

[1] Также относятся к итераторным адаптерам. — *Прим. ред.*

- ***ostream_iterator*** для записи в ***ostream*** (§3.4, §21.2.1).
- ***istream_iterator*** для чтения из ***istream*** (§3.6, §21.3.1).
- ***ostreambuf_iterator*** для записи в буфер потока (§21.6.1).
- ***istreambuf_iterator*** для чтения из буфера потока (§21.6.2).

Центральная идея состоит в том, чтобы представить ввод и вывод коллекций значений в виде последовательностей:

```
template<class T, class Ch = char, class Tr = char_traits<Ch> >
class ostream_iterator: public iterator<output_iterator_tag, void, void, void, void>
{
public:
  typedef Ch char_type;
  typedef Tr traits_type;
  typedef basic_ostream<Ch, Tr> ostream_type;

  ostream_iterator (ostream_type& s) ;
  ostream_iterator (ostream_type& s, const Ch* delim) ;
  ostream_iterator (const ostream_iterator&) ;
  ~ostream_iterator () ;

  ostream_iterator& operator= (const T& val) ;    // пишет val в поток вывода

  ostream_iterator& operator* () ;
  ostream_iterator& operator++ () ;
  ostream_iterator& operator++ (int) ;
};
```

Этот итератор берет обычные операции записи и инкремента итератора вывода и преобразует их в операцию << над ***ostream***. Например:

```
void f()
{
  ostream_iterator<int> os (cout) ;    // пишет int-ы в cout через os
  *os = 7;                             // выводим 7 (используя cout<<7)
  ++os;                                // готовимся к следующему выводу
  *os = 79;                            // выводим 79
}
```

Операция ++ или подготавливает реальную операцию вывода, или не делает ничего. Разные реализации могут осуществлять разные стратегии поведения, но в любом случае для переносимости кода операция ++ должна присутствовать между любыми двумя присваиваниями в ***ostream_iterator***. Естественно, все стандартные алгоритмы написаны именно таким образом — иначе они не стали бы работать с контейнером типа ***vector***. Все это объясняет причины, по которым ***ostream_iterator*** определен таким образом.

Реализация ***ostream_iterator*** тривиальна и оставляется в качестве упражнения (§19.6[4]). Стандартный ввод/вывод поддерживает разные типы символов; ***char_traits*** (§20.2) описывает аспекты символьных типов, важные для ввода/вывода и строк типа ***string***.

Потоковый итератор ввода для ***istream*** определяется аналогично:

```
template<class T, class Ch = char, class Tr = char_traits<Ch>, class Dist = ptrdiff_t>
class istream_iterator: public iterator<input_iterator_tag, T, Dist, const T*, const T&>
```

```
{
public:
  typedef Ch char_type;
  typedef Tr traits_type;
  typedef basic_istream<Ch, Tr> istream_type;

  istream_iterator();                          // конец ввода
  istream_iterator(istream_type& s);
  istream_iterator(const istream_iterator&);
  ~istream_iterator();

  const T& operator*() const;
  const T* operator->() const;
  istream_iterator& operator++();
  istream_iterator operator++(int);
};
```

Этот итератор определен таким образом, что обычные операции с контейнерами вызывают операцию >> над ***istream***. Например:

```
void f()
{
  istream_iterator<int> is(cin);    // читаем int-ы из cin через is
  int i1 = *is;                     // читаем int (используя cin>>i1)
  ++is;                             // готовимся к следующему вводу
  int i2 = *is;                     // читаем int
}
```

Умолчательный ***itsream_iterator*** обозначает конец ввода так, что мы можем явным образом задать входную последовательность:

```
void f(vector<int>& v)
{
  copy(istream_iterator<int>(cin), istream_iterator<int>(), back_inserter(v));
}
```

Чтобы это работало, стандартная библиотека предоставляет для типа ***istream_iterator*** операции == и !=.

Реализация ***istream_iterator*** уже не столь тривиальна, как реализация ***ostream_iterator***, но все равно не слишком сложна, так что я и ее оставляю в качестве упражнения (§19.6[5]).

19.2.6.1. Буфера потоков

Как объясняется в §21.6, потоки ввода/вывода основаны на идее о типах ***ostream*** и ***istream***, заполняющих и вычитывающих буфера, ввод/вывод в которые происходит на низком физическом уровне. Имеется возможность обойти форматирование стандартных потоков ввода/вывода и напрямую работать с их буферами (§21.6.4). Такая возможность предоставляется и алгоритмам через понятие итераторов ***istreambuf_iterator*** и ***ostreambuf_iterator***:

```
template<class Ch, class Tr = char_traits<Ch> >
class istreambuf_iterator:
        public iterator<input_iterator_tag, Ch, typename Tr::off_type, Ch*, Ch&>
```

```
{
public:
  typedef Ch char_type;
  typedef Tr traits_type;
  typedef typename Tr::int_type_int_type;
  typedef basic_streambuf<Ch, Tr> streambuf_type;
  typedef basic_istream<Ch, Tr> istream_type;

  class proxy;                                         // вспомогательный тип

  istreambuf_iterator() throw();                       // конец буфера
  istreambuf_iterator(istream_type& is) throw();       // читаем из streambuf объекта is
  istreambuf_iterator(streambuf_type*) throw();
  istreambuf_iterator(const proxy& p) throw();         // читаем из streambuf объекта p
  Ch operator*() const;
  istreambuf_iterator& operator++();
  proxy operator++(int);

  bool equal(istreambuf_iterator&);
};
```

Кроме того, предоставляются и операции == и !=.

Чтение из ***streambuf*** является более низкоуровневой операцией, чем из ***istream***. Как следствие, интерфейс ***istreambuf_iterator*** более хаотичен, чем у ***istream_iterator***. Тем не менее, при должной инициализации объектов типа ***istreambuf_iterator*** операции *, ++ и = обладают при традиционном использовании традиционной семантикой.

Тип ***proxy*** является вспомогательным типом, допускающим постфиксную операцию ++ без наложения ограничений на реализацию ***streambuf***. При инкрементировании итератора ***proxy*** хранит результирующее значение:

```
template<class Ch, class Tr = char_traits<Ch> >
class istreambuf_iterator<Ch, Tr>::proxy
{
  Ch val;
  basic_streambuf<Ch, Tr>* buf;

  proxy(Ch v, basic_streambuf<Ch, Tr>* b) : val(v), buf(b) { }

public:
  Ch operator*() {return val; }
};
```

Тип ***ostreambuf_iterator*** определяется аналогично:

```
template <class Ch, class Tr = char_traits<Ch> >
class ostreambuf_iterator: public iterator<output_iterator_tag, void, void, void, void>
{
public:
  typedef Ch char_type;
  typedef Tr traits_type;
  typedef basic_streambuf<Ch , Tr> streambuf_type;
  typedef basic_ostream<Ch , Tr> ostream_type;

  ostreambuf_iterator(ostream_type& os) throw();   // пишем в streambuf объекта os
  ostreambuf_iterator(streambuf_type*) throw();
  ostreambuf_iterator& operator=(Ch);
```

```
    ostreambuf_iterator& operator* () ;
    ostreambuf_iterator& operator++ () ;
    ostreambuf_iterator& operator++ (int) ;

    bool failed () const throw () ;          // true если Tr::eof() виден
};
```

19.3. Итераторы с проверкой

Кроме стандартных программист может определять и свои собственные итераторы. Часто это требуется при создании новых контейнеров, но бывает необходимым и при новом способе использования уже существующего контейнера. В качестве примера я здесь рассматриваю итератор, выполняющий проверку диапазона при доступе к элементам своего контейнера.

Применение стандартных контейнеров уменьшает необходимость в прямых манипуляциях с памятью. Применение стандартных алгоритмов снижает количество прямых обращений к элементам контейнера. Применение стандартной библиотеки вкупе со средствами языка C++ по строгой проверке типов резко снижают количество ошибок во время выполнения программ по сравнению с результатами, типичными для программирования в стиле языка C. Однако стандартная библиотека по-прежнему полагается на программиста в ситуациях потенциального выхода за границы диапазона элементов контейнера. Если по ошибке обратиться к элементу контейнера *x* как *x*[*x*.*size*()+*7*], то случится нечто непредвиденное и, обычно, малоприятное. Как мы знаем, здесь может в некоторых случаях помочь использование *векторов* с проверкой, таких как *Vec* (§3.7.2). В более общем контексте требуется применение итераторов с проверкой.

Чтобы на программиста не налагалось при этом слишком больших дополнительных нагрузок, нам требуется итератор с проверкой и удобным способом его прикрепления к контейнеру. Чтобы построить *Checked_iter*, нам нужны контейнер и итератор к нему. Я также предоставляю функции для удобного создания *Checked_iter* (как в случае «связывающих» адаптеров в §18.4.4.1, итераторов для вставок из §19.2.4 и т.д.):

```
template<class Cont, class Iter> Checked_iter<Cont,Iter> make_checked (Cont& c, Iter i)
{
  return Checked_iter<Cont,Iter> (c,i) ;
}

template<class Cont> Checked_iter<Cont,typename Cont::iterator> make_checked (Cont& c)
{
  return Checked_iter<Cont,typename Cont::iterator> (c,c.begin ()) ;
}
```

Эти функции удобны с точки зрения выведения типов из их фактических аргументов (явного объявления типов при этом не требуется). Например:

```
void f(vector<int>& v, const vector<int>& vc)
{
  typedef Checked_iter<vector<int>, vector<int>::iterator> CI;
  CI p1 = make_checked(v,v.begin()+3) ;
  CI p2 = make_checked(v) ;          // по умолчанию: указывает на первый эл-т
```

```
    typedef Checked_iter<const vector<int>, vector<int>::const_iterator> CIC;
    CIC p3 = make_checked(vc, vc.begin()+3);
    CIC p4 = make_checked(vc);

    const vector<int>& vv = v;
    CIC p5 = make_checked(v, vv.begin());
}
```

По умолчанию *константные* контейнеры имеют *константные* итераторы, из-за чего и их **Checked_iter** тоже константные. Итератор **p5** демонстрирует один из способов получения *константного* итератора из *неконстантного* контейнера. Это объясняет, почему **Checked_iter** нуждается в двух параметрах шаблона: один — для типа контейнера, другой — для выбора из альтернативы константный/неконстантный.

Имена типов для **Checked_iter** весьма длинны и запутаны, но это не имеет значения, когда итераторы используются в качестве аргументов стандартных алгоритмов. Например:

```
template<class Iter> void mysort(Iter first, Iter last);

void f(vector<int>& c)
{
  try
  {
    mysort(make_checked(c), make_checked(c, c.end()));
  }
  catch(out_of_bounds)
  {
    cerr << "oops: bug in mysort()\n";
    abort();
  }
}
```

Первоначальная версия этого алгоритма заставляла меня предположить, что тут весьма вероятны ошибки выхода за границу диапазона, так что применение здесь итераторов с проверкой явно имеет практический смысл.

В реализации **Checked_iter** имеем указатель на контейнер и итератор, привязанный к контейнеру:

```
template<class Cont, class Iter = typename Cont::iterator>
class Checked_iter: public iterator_traits<Iter>
{
  Iter curr;          // итератор для текущей позиции
  Cont* c;            // указатель на текущий контейнер
  // ...
};
```

Наследование от **iterator_traits** позволяет сразу иметь определения необходимых типов. Очевидная альтернатива — наследование от **iterator**, излишне многословна (как в случае **reverse_iterator** из §19.2.5). Но никакого требования, чтобы итераторы обязательно были классами, нет, а в случае классов нет требования, чтобы они наследовали от **iterator**.

Операции класса ***Checked_iter*** довольно просты:

```
template<class Cont, class Iher = typename Cont::iterator>
class Checked_iter: public iterator_traits<Iter>
{
  //...

public:
  void valid(Iter p) const
  {
    if(c->end() == p) return;
    for (Iter pp = c->begin(); pp!=c->end(); ++pp) if(pp == p) return;
    throw out_of_bounds();
  }

  friend bool operator==(const Checked_iter& i, const Checked_iter& j)
  {
    return i.c==j.c && i.curr==j.curr;
  }

  // Нет умолчательного инициализатора.
  Checked_iter(Cont& x, Iter p) : c(&x), curr(p) {valid(p);}
  Checked_iter(Cont& x) : c(&x), curr(x.begin()) {}

  reference operator*() const
  {
    if(curr==c->end()) throw out_of_bounds();
    return *curr;
  }

  pointer operator->() const
  {
    if(curr==c->end()) throw out_of_bounds();
    return &*curr;
  }

  Checked_iter operator+(difference_type d) const        // только для итераторов
                                                         // произвольного доступа
  {
    if(c->end()-curr<d || d<-(curr-c->begin())) throw out_of_bounds();
    return Checked_iter(c,curr+d);
  }

  reference operator[](difference_type d) const         // только для итераторов
                                                         // произвольного доступа
  {
    if(c->end()-curr<=d || d<-(curr-c->begin())) throw out_of_bounds();
    return curr[d];
  }

  Checked_iter& operator++()                             // префиксный ++
  {
    if(curr == c->end()) throw out_of_bounds();
    ++curr;
    return *this;
  }
```

```
    Checked_iter operator++ (int)                    // постфиксный ++
    {
        Checked_iter tmp = *this;
        ++*this;                                     // проверяется префиксным ++
        return tmp;
    }

    Checked_iter& operator-- ()                      // префиксный --
    {
        if(curr == c->begin ()) throw out_of_bounds () ;
        --curr;
        return *this;
    }

    Checked_iter operator-- (int)                    // постфиксный --
    {
        Checked_iter tmp = *this;
        --*this;                                     // проверяется префиксным --
        return tmp;
    }

    difference_type index () const {return curr-c->begin () ; }// только произвольный доступ

    Iter unchecked () const {return curr; }
    // +, -, <, и т.д. (§19.6[6])
};
```

Checked_iter может быть инициализирован только для конкретного итератора конкретного контейнера. В более полноценной версии для функции ***valid***() должна быть предоставлена более эффективная реализация, работающая с итераторами произвольного доступа (§19.6[6]). После инициализации ***Checked_iter*** при любых изменениях позиции итератора производится проверка, не выходит ли он при этом за границы диапазона элементов контейнера. Попытка заставить такой итератор с проверкой указывать за допустимые границы генерирует исключение ***out_of_bounds***. Например:

```
void f(list<string>& ls)
{
    int count = 0;

    try
    {
        Checked_iter<list<string> > p (ls, ls.begin ()) ;
        while (true)
        {
            ++p;            // рано или поздно достигнем конца
            ++count;
        }
    }
    catch (out_of_bounds)
    {
        cout<< "overrun after "<< count<< " tries\n";
    }
}
```

Checked_iter знает контейнер, к которому он привязан. Это позволяет ему отследить почти все случаи, когда итераторы контейнера становятся недействительными в результате выполнения над контейнером некоторых операций (§16.3.6, §16.3.8). Для полной защиты от таких случаев потребуется более сложный и дорогостоящий дизайн итератора с проверками (§19.6[7]).

Обратите внимание, что постфиксный инкремент ++ использует промежуточную переменную, а префиксный инкремент — нет. По этой причине для итераторов ++***p*** предпочтительнее, чем ***p***++.

Поскольку ***Checked_iter*** содержит указатель на контейнер, его невозможно напрямую применять ко встроенным массивам. При необходимости можно использовать ***c_array*** (§17.5.4).

В завершении темы итераторов с проверкой нам требуется найти способ их относительно простого использования. В этом отношении имеются два основных подхода:

1. Определить контейнер с проверкой, который ведет себя как остальные контейнеры, но предоставляет более ограниченный набор конструкторов, а его ***begin***() и ***end***() возвращают ***Checked_iter***, а не обычные итераторы.
2. Определить инициализируемый произвольным контейнером дескриптор, который и осуществляет проверку доступа к своему контейнеру (§19.6[8]).

Следующий шаблон прикрепляет итераторы с проверкой к контейнеру:

```
template<class C> class Checked: public C
{
public:
  explicit Checked(size_t n) : C(n) { }
  Checked() : C() { }

  typedef Checked_iter<C> iterator;
  typedef Checked_iter<C, C::const_iterator> const iterator;

  iterator begin() {return iterator(*this,C::begin());}
  iterator end() {return iterator(*this, C::end());}
  const_iterator begin() const {return const_iterator(*this,C::begin());}
  const_iterator end() const {return const_iterator(*this,C::end());}

  typename C::reference_type operator[] (typename C::size_type n)
  {return Checked_iter<C>(*this)[n];}

  C& base() {return *this;}           // получаем базовый контейнер
};
```

Теперь мы можем написать:

```
Checked<vector<int> > vec(10);
Checked<list<double> > lst;

void f()
{
  int i1 = vec[5];                    // ok
  int i2 = vec[15];                   // генерируется out_of_bounds
  // ...
  mysort(vec.begin(),vec.end());
  copy(vec.begin(),vec.end(),lst.begin());
}
```

Очевидным образом избыточная функция *base*() призвана обеспечивать типичный для дескрипторов интерфейс. Дескрипторы контейнеров обычно не обеспечивают неявного приведения от типа дескриптора к типу контейнера.

Если изменяется размер контейнера, все его итераторы, в том числе и итераторы с проверкой, могут стать недействительными. В таких случаях можно заново проинициализировать *Checked_iter*:

```
void g (vector<int>& vi)
{
  Checked_iter<vector<int> > p (vi) ;
  //..
  int i = p.index () ;                              // получаем текущую позицию
  vi.resize (100) ;                                 // p становится недействительным
  p = Checked_iter<vector<int> > (vi, vi.begin () +i) ; // восстанавливаем текущую позицию
}
```

Старая (и недействительная) текущая позиция теряется. Я предоставил функцию *index*() как средство извлечения индекса, что позволяет восстановить *Checked_iter*.

19.3.1. Исключения, контейнеры и алгоритмы

Вы можете возразить, что одновременное применение стандартных алгоритмов и итераторов с проверкой сродни одновременному использованию ремня и подтяжек. Однако опыт показывает, что для некоторых людей и некоторых приложений небольшая доза паранойи оправдана, особенно если в программу вносят изменения разные люди.

Один из способов контроля ошибок времени выполнения состоит во внесении в код специальных проверочных фрагментов. Под самый конец, когда программа отлажена, они удаляются. Такую практику можно сравнить с надеванием спасательного жилета при плавании вдоль берега и избавлении от него при выходе в открытое море. Но и всегда оставлять проверочные фрагменты нереалистично, так как многие из них действительно заметно снижают эффективность выполнения программы. Поэтому неплохо произвести контрольные замеры и посмотреть, что именно заметно тормозит выполнение программы. И после этаких замеров отбросить лишь эти проверочные фрагменты, причем из наиболее тщательно протестированных участков кода программы, а остальные — оставить.

Применение *Checked_iter* позволяет выявить многие ошибки. Это не означает, что от них легко избавиться. Редко когда удается написать программу, на 100% устойчивую к операциям ++, --, *, [], -> и =, потенциально генерирующим исключения. У нас имеются в связи с этим две очевидные стратегии:

1. Перехватывать исключения как можно ближе к местам их возникновения, чтобы обработчик исключения имел неплохой шанс разузнать все о причинах генерации исключения.
2. Перехватывать исключения на верхних уровнях программы и прерывать тем самым работу изрядных кусков программы (оставляя под подозрением все структуры данных, попадающих в указанные части программы).

Просто перехватывать исключения, возникшие в неизвестной части программы, и продолжать вычисления в надежде на то, что все структуры данных остались

в действительных состояниях, безответственно если нет обработки ошибок на более верхних уровнях программы. Простейший пример — финальная проверка результатов вычислений (машиной или человеком). В таких случаях проще и дешевле продолжить работу, чем безнадежно пытаться отловить на нижнем уровне все мыслимые и немыслимые ошибки. Это упрощение возможно за счет многоуровневой схемы восстановления после возникновения ошибок (§14.9).

19.4. Аллокаторы (распределители памяти)

Аллокаторы (*allocators* — распределители памяти) призваны изолировать разработчика алгоритмов и контейнеров, нуждающихся в выделении памяти, от низкоуровневых деталей физической организации памяти. Аллокаторы предоставляют стандартные способы выделения и освобождения памяти, а также стандартные имена типов, используемых для указателей и ссылок. Аллокаторы, как и итераторы, суть чистые абстракции. Любой тип, ведущий себя как аллокатор, и есть аллокатор.

Стандартная библиотека предоставляет стандартный аллокатор, призванный удовлетворить типичные нужды большинства пользователей. Но пользователь может разрабатывать и свои собственные аллокаторы, чтобы реализовать альтернативные схемы работы с памятью. Например, речь может идти об аллокаторах, работающих с разделяемой памятью, памятью с автоматической сборкой мусора, памятью на базе заранее выделенного пула объектов (§19.4.2) и т.д.

Стандартные контейнеры и алгоритмы работают с памятью посредством аллокатора. Таким образом, предоставляя новый аллокатор, мы предоставляем стандартным контейнерам возможность по-новому работать с памятью.

19.4.1. Стандартный аллокатор

Стандартный шаблон ***allocator*** из заголовочного файла <***memory***> выделяет память посредством функции ***operator new***() (§6.2.6), и по умолчанию используется всеми стандартными контейнерами:

```
template<class T> class std::allocator
{
public:
  typedef T value_type;
  typedef size_t size_type;
  typedef ptrdiff_t difference_type;
  typedef T* pointer;

  typedef const T* const_pointer;

  typedef T& reference ;
  typedef const T& const_reference;

  pointer address(reference r) const {return &r; }
  const_pointer address(const_reference r) const {return &r; }

  allocator() throw() ;
  template<class U> allocator(const allocator<U>&) throw() ;
  ~allocator() throw() ;
```

```
    pointer allocate (size_type n, allocator<void>::const_pointer hint=0); // память для n Ts
    void deallocate (pointer p, size_type n); // освобождает память без уничтож-я объектов

    void construct (pointer p, const T& val) {new (p) T(val);} // инициал-я *p значением val
    void destroy (pointer p) {p->~T();}        // уничтожает *p без освобождения памяти

    size_type max_size() const throw();

    template<class U>
    struct rebind {typedef allocator<U> other;}; // фактически: typedef allocator<U> other
};

template<class T> bool operator==(const allocator<T>&, const allocator<T>&) throw();
template<class T> bool operator!=(const allocator<T>&, const allocator<T>&) throw();
```

Операция ***allocate*(*n*)** выделяет память для ***n*** объектов, а для ее освобождения нужно вызвать ***deallocate*(*p*, *n*)**. Заметьте, что ***deallocate*()** также принимает в качестве аргумента число объектов ***n***. Это позволяет аллокаторам, близким по оптимальности к эффективным, ограничиться хранением минимальной информации о выделяемой ими памяти. С другой стороны, такие аллокаторы требуют от пользователя при вызове ***deallocate*()** всегда предоставлять правильное ***n***. Заметим, что у функции ***deallocate*()**, в отличие от функции operator ***delete*()** (§6.2.6), аргумент не может быть равен *нулю*.

Умолчательная реализация класса ***allocator*** использует ***operator new*(*size_t*)** для выделения памяти и ***operator delete*(*void**)** для ее освобождения. Это подразумевает, что может быть вызвана ***new_handler*()** и в случае исчерпания памяти может генерироваться исключение ***std*::*bad_alloc*** (§6.2.6.2).

Отметим, что ***allocate*()** не обязана вызывать исключительно низкоуровневые механизмы работы с памятью. Часто аллокатору разумнее хранить список свободных областей памяти, готовых к выделению за минимальное время (§19.4.2).

Необязательный аргумент ***hint*** у функции ***allocate*()** является специфическим для разных реализаций. Например, он полезен для систем, у которых компактность крайне важна. Например, аллокатор может выделять память для связанных объектов в пределах одной и той же страницы в случае страничной организации памяти. Типом аргумента ***hint*** служит ***pointer*** из сверхупрощенной специализации:

```
template<> class allocator<void>
{
public:
    typedef void* pointer;
    typedef const void* const_pointer;

    // внимание: отсутствует reference

    typedef void value_type;

    template<class U>
    struct rebind {typedef allocator<U> other;}; // фактически: typedef allocator<U> other
};
```

Тип ***allocator*<*void*>::*pointer*** действует как тип универсального указателя и для стандартных аллокаторов есть просто ***void****.

Если только в документации на аллокатор не сказано иного, у пользователя есть следующий выбор при вызове ***allocate***():

1. Не предоставлять аргумент ***hint***.
2. Использовать в его качестве указатель на объект, часто применяемый в связи с новым объектом (например, указатель на предыдущий элемент последовательности).

Аллокаторы избавляют разработчиков контейнеров от необходимости работать напрямую с «*сырой памятью*» (*raw memory*). Для примера рассмотрим, как реализация ***vector*** могла бы работать с памятью:

```
template<class T, class A = allocator<T> > class vector
{
public:
  typedef typename A::pointer iterator;
  // ...
private:
  A alloc;                        // объект аллокатора
  iterator v;                     // указатель на элементы
  // ...
public:
  explicit vector(size_type n, const T& val = T(), const A& a = A())
          : alloc(a)
  {
    v = alloc.allocate(n);
    for(iterator p = v; p<v+n; ++p) alloc.construct(p,val);
    // ...
  }

  void reserve(size_type n)
  {
    if(n<=capacity()) return;

    iterator p = alloc.allocate(n);
    iterator q = v;

    while(q<v+size())             // копируем существующие элементы
    {
      alloc.construct(p++,*q);
      alloc.destroy(q++);
    }

    alloc.deallocate(v,capacity()); // освобождаем старую память
    v = p - size();
    // ...
  }
  // ...
};
```

Операции аллокатора выражаются в терминах определенных с помощью ***typedef*** типов ***pointer*** и ***reference***, чтобы предоставить пользователю возможность использования альтернативных типов для доступа к памяти. В общем случае это сделать непро-

сто. Например, невозможно средствами языка C++ предоставить абсолютно совершенный ссылочный тип. Однако разработчики языка и библиотек могут воспользоваться этими *typedef* для поддержки типов, которые не могут быть предоставлены рядовым пользователем. Например, речь может идти об аллокаторе, предоставляющем доступ к долговременной памяти. Или о «длинных указателях», предоставляющих доступ к памяти за пределами обычного 32-адресного пространства.

Обычный пользователь может ввести тип необычного указателя для аллокатора, служащего каким-то специфическим нуждам. Этого, однако, нельзя сделать для ссылок.

Аллокаторы упрощают работу с объектами типа, специфицированного параметром шаблона. Однако большинство контейнеров работает и с объектами иных типов. Например, разработчику контейнера *list* требуется размещать в памяти объекты типа *Link*. И обычно их нужно размещать в памяти аллокатором контейнера *list*.

Любопытный тип *rebind* введен для того, чтобы позволить аллокаторам размещать в памяти объекты произвольных типов. Рассмотрим следующий оператор *typedef*:

```
typedef typename A::template rebind<Link>::other Link_alloc; // "template;" см. §C.13.6
```

Если *A* — это аллокатор типа *allocator*, то тогда *rebind*<*Link*>::*other* означает *allocator*<*Link*>, и рассмотренный оператор *typedef* есть просто косвенный способ сказать следующее:

```
typedef allocator<Link> Link_alloc;
```

Косвенность в данном случае освобождает нас от необходимости явного упоминания типа *allocator*. Тип *Link_alloc* выражается в терминах параметра шаблона *A*. Например:

```
template<class T, class A = allocator<T> > class list
{
private:
  class Link { /* ... */ };

  typedef typename A::template rebind<Link>::other Link_alloc;   // allocator<Link>

  Link_alloc a;                          // аллокатор для link
  A alloc;                               // аллокатор для list
  // ...

public:
  typedef typename A::pointer iterator;
  // ...
  iterator insert(iterator position, const T& x)
  {
    Link_alloc::pointer p = a.allocate(1);  // получить Link
    // ...
  }
  // ...
};
```

Поскольку *Link* — это член класса *list*, он также параметризуется аллокатором (§13.2.1). Как следствие, типы *Link* для контейнеров *link* с разными аллокаторами тоже разные, как и типы самих контейнеров (§17.3.3).

19.4.2. Пользовательский аллокатор

Часто разработчики контейнеров применяют операции ***alocate***() и ***deallocate***() по одному разу на каждый объект. Для простейшей реализации ***allocate***() это означает массу вызовов функции operator ***new***(), а реализации последней не так уж и эффективны в этом случае. В качестве примера пользовательского аллокатора я применяю схему с заранее сформированным пулом участков памяти фиксированного размера, из которого можно будет выделять память под объекты более эффективно, чем это делает стандартная функция общего назначения operator ***new***().

В свое время я столкнулся с распределителем памяти из заранее сформированного пула, который работал правильно, но имел неправильный интерфейс (так как был разработан задолго до изобретения аллокаторов). Это был класс ***Pool***, который и формировал понятие пула элементов фиксированного размера, из которого пользователь мог быстро выделять память под объекты и освобождать ее. Это был низкоуровневый тип, работавший с памятью напрямую и учитывавший необходимость в выравнивании границ памяти:

```
class Pool
{
  struct Link { Link* next; };

  struct Chunk
  {
    enum {size = 8*1024-16};  // слегка меньше 8K, чтобы кусок памяти умещался в 8K
    char mem[size];           // для достижения точного выравнивания
    Chunk* next;
  };

  Chunk* chunks;
  cost unsigned int esize;
  Link* head;
  Pool(Pool&);                // защита от копирования
  void operator=(Pool&);      // защита от копирования
  void grow();                // увеличить пул

public:
  Pool(unsigned int n);       // n это размер элементов
  ~Pool();

  void* alloc();              // выделить память под один элемент
  void free(void* b);         // возвращение элемента в пул
};

inline void* Pool::alloc()
{
  if(head==0) grow();
  Link* p = head;             // вернуть первый элемент
  head = p->next;
  return p;
}

inline void Pool::free(void* b)
{
  Link* p = static_cast<Link*>(b);
```

```
    p->next = head;
    head = p;
}

Pool::Pool(unsigned int sz)
    : esize(sz<sizeof(Link)?sizeof(Link):sz)
{
    head = 0;
    chunks = 0;
}

Pool::~Pool()                   // освободить все куски (chunks)
{
    Chunk* n = chunks;
    while(n)
    {
        Chunk* p = n;
        n = n->next;
        delete p;
    }
}

void Pool::grow()               // выделяет новый 'chunk,' организуя его в виде связанного
                                // списка элементов размером 'esize'
{
    Chunk* n = new Chunk;
    n->next = chunks;
    chunks = n;

    const int nelem = Chunk::size/esize;
    char* start = n->mem;
    char* last = &start[(nelem-1)*esize];

    for(char* p = start; p<last; p+=esize)
        reinterpret_cast<Link*>(p)->next = reinterpret_cast<Link*>(p+esize);

    reinterpret_cast<Link*>(last)->next = 0;
    head = reinterpret_cast<Link*>(start);
}
```

Чтобы продемонстрировать чуть-чуть реализма, я буду использовать ***Pool*** в неизменном виде как часть моего собственного аллокатора (вместо того, чтобы исправлять его интерфейс). Мой аллокатор на базе фиксированного пула призван быстро выделять и освобождать память под один элемент, и именно это делает класс ***Pool***. Расширение данной модели на более общие случаи выделения памяти под несколько объектов или под объекты разных размеров (как того требует ***rebind***) я оставляю в качестве упражнения (§19.6[9]).

При наличии ***Pool*** определение ***Pool_alloc*** тривиально:

```
template <class T> class Pool_alloc
{
private:
    static Pool mem;            // пул элементов размером sizeof(T)

public:
    // аналогично стандартному allocator (§19.4.1)
};
```

```
template<class T> Pool Pool_alloc<T>::mem(sizeof(T));

template<class T> Pool_alloc<T>::Pool_alloc() { }

template<class T>
T* Pool_alloc<T>::allocate(size_type n, void* = 0)
{
  if(n == 1) return static_cast<T*>(mem.alloc());
  //...
}

template<class T>
void Pool_alloc<T>::deallocate(pointer p, size_type n)
{
  if(n == 1)
  {
    mem.free(p);
    return;
  }
  //...
}
```

Применение аллокатора также очевидно:

```
vector<int, Pool_alloc<int> > v;
map<string, number, Pool_alloc<pair<const string, number> > > m;

// используем как обычно
vector<int> v2 = v;                // error: иные параметры аллокатора
```

Я предпочел сделать поле типа ***Pool*** статическим в классе ***Pool_alloc*** из-за ограничений, которые стандартная библиотека накладывает на аллокаторы стандартных контейнеров: реализации стандартных контейнеров могут обращаться с объектами своего аллокаторного типа как с эквивалентными объектами. Это существенно повышает эффективность реализации. Из-за этого ограничения, например, не нужно выделять память под аллокаторы объектов типа ***Link*** (в типичных случаях они параметризуются аллокаторами своих контейнерных классов; §19.4.1), и операции над элементами двух последовательностей (например, ***swap***()) могут не проверять, одинаковые ли аллокаторы обслуживают элементы обоих последовательностей. Указанное ограничение, однако, не позволяет аллокаторам иметь доступ к данным конкретных экземпляров объектов.

Перед тем, как разрабатывать или применять пользовательский аллокатор, убедитесь в его реальной необходимости. Я предполагаю, что многие стандартные аллокаторы могут предоставлять классические оптимизационные схемы, а в таком случае нет нужды в собственных разработках.

19.4.3. Обобщенные аллокаторы

Класс ***allocator*** представляет собой простой оптимизирующий вариант идеи о передаче информации контейнеру через параметр шаблона (§13.4.1, §16.2.3). Например, имеет смысл требовать, чтобы память под каждый элемент контейнера выделялась исключительно через аллокатор. В то же время, если разрешить двум спискам типа ***list*** иметь разные аллокаторы, то в таком случае ***splice***() (§17.2.2.1) нельзя

было бы реализовывать обменом ссылок; пришлось бы применять поэлементное копирование и все по причине учета такого редкого случая, когда **splice**() действует над элементами последовательностей с разными аллокаторами одного и того же аллокаторного типа. Аналогично, если сделать аллокаторы совершенно универсальными, то механизмы вроде ***rebind***, позволяющие аллокаторам выделять память под элементы произвольных типов, стали бы слишком сложными. Как следствие, стандартным аллокаторам запрещается работать с данными объектов, и реализации стандартных контейнеров могут извлекать из этого выгоду.

Удивительно, но кажущееся драконовским ограничение на пообъектную информацию в аллокаторах не слишком серьезно. Большинство аллокаторов не нуждается в пообъектных данных и могут работать быстрее без такой информации. Однако аллокаторы могут хранить данные на уровне типа аллокатора. Если нужны разные данные, можно воспользоваться разными типами аллокаторов. Например:

```
template<class T, class D> class My_alloc // allocator для T с реализацией посредством D
{
  D d;                                    // нужно для My_alloc<T,D>
  // ...
};

typedef My_alloc<int, Persistent_info> Persistent;
typedef My_alloc<int, Shared_info> Shared;
typedef My_alloc<int, Default_info> Default;

list<int, Persistent> lst1;
list<int, Shared> lst2;
list<int, Default> lst3;
```

Списки ***lst1***, ***lst2*** и ***lst3*** относятся к разным типам. Поэтому при работе с двумя из этих списков мы должны использовать универсальные алгоритмы (глава 18), а не специализированные операции класса списков (§17.2.2.1). В итоге, серьезных проблем при использовании различных аллокаторов не возникает — просто выполняется копирование элементов, а не переназначение ссылок.

Ограничение на пообъектную информацию в аллокаторах накладывается в первую очередь из-за строжайших требований к высочайшей эффективности стандартной библиотеки. К примеру, расходы памяти под аллокаторные данные списков невелики, но они сильно возрастают, если хранить данные о каждой связи.

Теперь рассмотрим, как можно было бы применить аллокаторную технологию, если бы не было строжайших ограничений на эффективность работы. Это актуально либо для нестандартных библиотек, от которых не требуется высокой эффективности для любой структуры данных и любого типа программы, либо для некоторых специфических реализаций стандартной библиотеки. В этих случаях аллокаторы могли бы содержать информацию, типичную для универсальных базовых классов (§16.2.2). Например, можно было бы разработать аллокатор так, чтобы он мог отвечать на вопросы: «где расположены объекты?», «какова их структура?», «содержится ли объект в контейнере?» и т.д. Он мог бы предоставлять сервисы контейнерам, действующим как кэш объектов постоянной памяти, мог бы обеспечивать ассоциативные связи между контейнерами и иными объектами и т.п.

Таким прозрачным способом можно было бы снабдить произвольными услугами обычные контейнерные операции. Однако лучше всего четко разграничивать

вопросы хранения данных и вопросы использования данных. Последние не соответствуют обобщенным аллокаторам, и их лучше предоставлять через отдельные параметры шаблона.

19.4.4. Неинициализированная память

Помимо стандартного аллокатора (шаблон ***allocator***) заголовочный файл <***memory***> определяет несколько функций для работы с неинициализированной памятью. Они обладают опасным, но порой очень важным свойством — использовать имя типа *T*, чтобы работать не с корректно сконструированным объектом типа *T*, а с областью памяти, достаточной для хранения объекта типа *T*.

Библиотека предоставляет три способа копировать значения в неинициализированную память:

```
template<class In, class For>
For uninitialized_copy(In first, In last, For res)
{
  typedef typename iterator_traits<For>::value_type V;

  while (first != last)
    new (static_cast<void*>(&*res++)) V(*first++);    // см. (§10.4.11)
  return res;
}

template<class For, class T>
void uninitialized_fill(For first, For last, const T& val)
{
  typedef typename iterator_traits<For>::value_type V;

  while (first != last)
    new (static_cast<void*>(&*first++)) V(val);
}

template<class For, class Size, class T>
void uninitialized_fill_n(For first, Size n, const T& val)
{
  typedef typename iterator_traits<For>::value_type V;

  while (n--)
    new (static_cast<void*>(&*first++)) V(val);
}
```

Эти функции предназначены в первую очередь для разработчиков контейнеров и алгоритмов. Например, ***reserve***() и ***resize***() (§16.3.8) легче всего реализуются с помощью этих функций (§19.6[10]). Но будет абсолютно неприемлемо, если подобного рода «недоинициализированный объект» выйдет за пределы внутренней реализации контейнера и попадет в руки обычного, конечного пользователя. См. также §E.4.4.

Часто алгоритмы нуждаются в промежуточной памяти ради приемлемой производительности. И часто такую память лучше всего выделить за одну операцию без соответствующей инициализации, оставив таковую вплоть до момента, когда реально в работу потребуется некоторый конкретной кусок такой памяти. В итоге, библиотека предоставляет две функции для выделения и освобождения неинициализированной памяти:

```
// выделить, но не инициализировать
template<class T> pair<T*,ptrdiff_t> get_temporary_buffer(ptrdiff_t);

// освободить, но не уничтожать
template<class T> void return_temporary_buffer(T*);
```

Функция ***get_temporary_buffer<X>(n)*** пытается выделить память, достаточную для размещения n или более объектов типа *X*. Если ей удается выделить некоторое количество памяти, то она возвращает указатель на начало выделенной неинициализированной памяти и число объектов типа *X*, которые могут там разместиться; в противном случае поле ***second*** в возвращаемой паре равно нулю. Идея заключается в том, что сама система следит за числом готовых к быстрому выделению блоков памяти заданного размера, так что выделяемый блок может вмещать и более ***n*** запрошенных объектов. Но может быть выделен и блок размера, в котором помещается меньше, чем n объектов, так что применение функции ***get_temporary_buffer()*** состоит в том, что мы просим побольше, а получаем сколько дадут.

Память, выделенную функцией ***get_temporary_buffer()***, нужно для ее дальнейшего использования освобождать функцией ***return_temporary_buffer()***. Аналогично тому, что ***get_temporary_buffer()*** выделяет память без ее инициализации, функция ***return_temporary_buffer()*** освобождает память без соответствующего уничтожения объектов. Поскольку ***get_temporary_buffer()*** является низкоуровневой функцией и скорее всего оптимизирована для работы с временными буферами, ее не следует использовать вместо операции ***new*** или функции ***allocator::allocate()*** для получения долговременного хранилища.

Стандартные алгоритмы, осуществляющие запись в последовательности элементов, рассчитывают на то, что элементы загодя проинициализированы. То есть алгоритмы применяют присваивания, а не копирующие конструкторы. Как следствие, мы не можем непосредственно применить неинициализированную память для работы алгоритмов. Все это неприятно, ибо присваивание может оказаться значительно дороже инициализации. Кроме того, нас мало интересуют значения, которые мы будем переписывать (иначе мы бы не стали их переписывать). Решение проблемы заключается в использовании класса ***raw_storage_iterator*** из заголовочного файла <***memory***>, который выполняет инициализацию, а не присваивание:

```
template<class Out, class T>
class raw_storage_iterator: public iterator<output_iterator_tag,void,void,void,void>
{
  Out p;

public:
  explicit raw_storage_iterator(Out pp) : p(pp) { }

  raw_storage_iterator& operator*() {return *this;}
  raw_storage_iterator& operator=(const T& val)
  {
    T* pp = &*p;
    new (pp) T(val);      // размещаем val в pp (§10.4.11)
    return *this;
  }

  raw_storage_iterator& operator++() {++p; return *this;}
  raw_storage_iterator operator++(int)
```

```
  {
    raw_storage_iterator t = *this;
    ++p;
    return t;
  }
};
```

Напишем, например, шаблонную функцию, которая копирует содержимое контейнера типа *vector* в буфер:

```
template<class T, class A> T* temporary_dup (vector<T,A>& v)
{
  pair<T*,ptrdiff_t> p = get_temporary_buffer<T> (v.size ());

  if(p.second < v.size ())        // проверка наличия доступной памяти
  {
    if(p.first != 0) return_temporary_buffer (p.first);
    return 0;
  }

  copy (v.begin (), v.end (), raw_storage_iterator<T*,T> (p.first));
  return p.first;
}
```

Если бы вместо *get_temporary_buffer()* применялась операция *new*, то инициализация была бы выполнена. А поскольку мы избежали инициализации, требуется применять *raw_storage_iterator* для работы с неинициализированной памятью. Наконец, в данном примере клиентский код, вызывающий *temporary_dup()*, отвечает за вызов *return_temporary_buffer()* с указателем, который он (клиент) получает в качестве возврата.

19.4.5. Динамическая память

Средства, применяемые для реализации операций *new* и *delete*, объявлены в заголовочном файле <*new*>:

```
class bad_alloc: public exception { /* ... */ };

struct nothrow_t { };
extern const nothrow_t nothrow;   // индикатор выделения памяти, не генерир-го исключений

typedef void (*new_handler) ();
new_handler set_new_handler (new_handler new_p) throw (); // вернуть старый new_handler

void* operator new (size_t) throw (bad_alloc);
void operator delete (void*) throw ();

void* operator new (size_t, const nothrow_t&) throw ();
void operator delete (void*, const nothrow_t&) throw ();

void* operator new[] (size_t) throw (bad_alloc);
void operator delete[] (void*) throw ();

void* operator new[] (size_t, const nothrow_t&) throw ();
void operator delete[] (void*, const nothrow_t&) throw ();
```

```
void* operator new (size_t, void* p) throw () {return p; }     // размещение (§10.4.11)
void operator delete (void* p, void*) throw () { }             // не делаем ничего

void* operator new [] (size_t, void* p) throw () {return p; }
void operator delete [] (void* p, void*) throw () { }          // не делаем ничего
```

Функции ***operator new***() и ***operator new***[] () с пустыми спецификациями исключений (§14.6) не могут сигнализировать об исчерпании памяти генерацией исключения ***std*::*bad_alloc***. Вместо этого при неудачной попытке выделения памяти они возвращают *нуль*. В выражении, содержащем операцию ***new*** (§6.2.6.2), производится проверка возврата от средства выделения памяти с пустой спецификацией исключений; если этот возврат равен нулю, никакого конструктора не вызывается и все выражение также возвращает *нуль*. В частности, все функции с ***nothrow*** в случае неудачи возвращают нуль и не пытаются генерировать исключение ***bad_alloc***. Например:

```
void f()
{
  int* p = new int[100000] ;                  // может сгенерировать bad_alloc
  if(int* q = new (nothrow) int[100000] )     // не генерирует исключений
  {
    // выделение памяти успешно
  }
  else
  {
    // выделение памяти неуспешно
  }
}
```

Это позволяет применить средства обработки ошибок динамического выделения памяти без генерации исключений.

19.4.6. Выделение памяти в стиле языка С

От языка С язык С++ унаследовал функциональный интерфейс для работы с динамической памятью. Его можно найти в заголовочном файле <***cstdlib***>:

```
void* malloc (size_t s) ;              // выделяет s байт
void* calloc (size_t n, size_t s) ;    // выделяет n раз по s байт (инициализированных 0)
void free (void* p) ;                  // освобождает память, выделенную malloc() или calloc()
void* realloc (void* p, size_t s) ;    // изменяет размер массива, указуемого через p, до s;
```

Этих функций следует избегать и предпочитать им операции ***new***, ***delete*** и стандартные контейнеры. Эти функции работают с неинициализированной памятью. В частности, функция ***free***() не вызывает деструкторы для объектов в памяти, которую она освобождает. Реализации операций ***new*** и ***delete*** могут использовать эти функции, но могут и не использовать. Выделение памяти операцией new и ее освобождение функцией ***free***() — гарантированный способ нарваться на неприятности. Если вы испытываете потребность в функции ***realloc***(), лучше подумайте, как вам использовать стандартный контейнер; это и проще, и не менее эффективно (§16.3.5).

В библиотеке имеется также группа функций, предназначенная для эффективного манипулирования байтами. Поскольку язык С изначально обращался к безтиповым байтам по указателям типа ***char****, функции эти сосредоточены в <***cstring***>. В пределах этих функций указатели ***void**** трактуются как ***char****:

```
void* memcpy(void* p, const void* q, size_t n);  // копир-е неперекрыв-ся обл-ей
void* memmove(void* p, const void* q, size_t n); // копир-е потенц-но перекрыв-ся обл-ей
```

Подобно ***strcpy*()** (§20.4.1) эти функции копируют ***n*** байт из ***q*** в ***p*** и возвращают ***p***. Диапазоны (интервалы), копируемые функцией ***memmove*()**, могут перекрываться. В то же время, функция ***memcpy*()** не допускает перекрытия интервалов, и это допущение позволяет оптимизировать ее код. Аналогично:

```
// как strchr() (§20.4.1): находит b в p[0]..p[n-1]
void* memchr(const void* p, int b, size_t n);

// как strcmp(): сравнивает последовательности байтов
int memcmp(const void* p, const void* q, size_t n);

void* memset(void* p, int b, size_t n);  // устанавливает n байтов в b, возвращает p
```

Многие реализации предоставляют высокооптимизированные версии этих функций.

19.5. Советы

1. При написании алгоритмов решите, какой вид итераторов выбрать для обеспечения требуемой эффективности, и далее применяйте только операции над этим типом итераторов; §19.2.1.
2. Для обеспечения эффективных реализаций алгоритмов с аргументами-итераторами, предоставляющими расширенный сервис, используйте перегрузку; §19.2.3.
3. Используйте ***iterator_traits*** для алгоритмов, соответствующих разным категориям итераторов; §19.2.2.
4. Не забывайте использовать операцию ++ между обращениями к ***istream_iterator*** и ***ostream_iterator***; §19.2.6.
5. Во избежание переполнения контейнеров применяйте итераторы вставок; §18.3, §19.2.4.
6. Применяйте дополнительный проверочный код во время отладки, и выкидывайте его в дальнейшем лишь по необходимости; §19.3.1.
7. Вместо ***p*++** используйте **++*p***; §19.3.
8. Для повышения эффективности алгоритмов, работающих с нестандартными структурами данных, пользуйтесь неинициализированной памятью; §19.4.4.
9. Применяйте временные буферы для повышения эффективности алгоритмов, использующих временные структуры данных; §19.4.4.
10. Хорошенько подумайте, прежде чем писать свой собственный аллокатор; §19.4.
11. Избегайте применения функций ***malloc*()**, ***free*()**, ***realloc*()** и т.д.; §19.4.6.
12. Применяя технику ***rebind***, вы можете сымитировать ***typedef*** с шаблоном; §19.4.1.

19.6. Упражнения

1. (*1.5) Реализуйте ***reverse***() из §18.6.7. Подсказка: см. §19.2.3.
2. (*1.5) Напишите итератор вывода, ***Sink***, который в действительности ничего никуда не пишет. Где такой ***Sink*** может быть полезен?
3. (*2) Реализуйте ***reverse_iterator*** (§19.2.5).
4. (*1.5) Реализуйте ***ostream_iterator*** (§19.2.6).
5. (*2) Реализуйте ***istream_iterator*** (§19.2.6).
6. (*2.5) Завершите ***Checked_iter*** (§19.3).
7. (*2.5) Переделайте ***Checked_iter*** так, чтобы проверять недействительные итераторы.
8. (*2) Спроектируйте и реализуйте дескрипторный класс, предоставляющий прокси-интерфейс к контейнеру. Его реализация должна базироваться на указателе на контейнер плюс реализация контейнерных операций с проверкой диапазона.
9. (*2.5) Завершите или реализуйте с самого начала ***Pool_alloc*** (§19.4.2) так, чтобы обеспечивались все возможности стандартного аллокатора ***allocator*** (§19.4.1) из стандартной библиотеки. Сравните производительности ***Pool_alloc*** и ***allocator,*** чтобы решить, стоит ли использовать ***Pool_alloc*** в вашей системе.
10. (*2.5) Реализуйте ***vector***, применяя аллокаторы, а не операции ***new*** и ***delete***.

20

Строки

Предпочитайте стандарт всему остальному.
— Странк и Уайт

Строки — символы — шаблон ***char_traits*** — стандартный шаблон ***basic_string*** — итераторы — доступ к элементам (символам) — конструкторы — обработка ошибок — присваивание — преобразования — сравнения — вставка — конкатенация— поиск и замена — размер и емкость — ввод/вывод строк — С-строки — классификация символов — функции из библиотеки С — советы — упражнения.

20.1. Введение

Строка — это последовательность символов. Стандартный библиотечный класс ***string*** обеспечивает такие операции со строками, как индексация (§20.3.3), присваивание (§20.3.6), сравнение (§20.3.8), добавление (appending; §20.3.9), конкатенация (§20.3.10) и поиск подстрок (§20.3.11). Для подстрок стандарт не предусматривает никаких общих средств, а то, что представлено в данной главе, следует рассматривать в первую очередь как пример работы со строками (§20.3.13). Стандартные строки могут состоять из символов практически любого типа (§20.2).

Опыт показывает, что невозможно разработать совершенный строковый класс. Вкусы людей по этому поводу, их привычки и ожидания различаются слишком сильно. Так что стандартный библиотечный класс ***string*** вовсе не идеален. Я бы кое-что изменил в нем, и у вас нашлись бы свои предпочтения. Тем не менее, во многих случаях этот класс работает хорошо, к нему легко добавляются функции, отвечающие специфическим нуждам, и, наконец, класс ***std::string*** хорошо известен и легко доступен. По большому счету, эти достоинства перевешивают те небольшие улучшения, которые мы могли бы в нем произвести. На образовательной ниве прижились многочисленные эксперименты по написанию кода строковых классов (§11.12, §13.2), но в реальной практике следует применять именно библиотечный класс ***string***.

Языку C++ в наследство от языка C достались строки, понимаемые как массив символов типа ***char*** с терминальным нулем, а также набор функций для работы с такими C-строками (§20.4.1).

20.2. Символы

«Символы» («characters») сами по себе — это интересная концепция. Рассмотрим символ *C*. Его можно представлять себе в виде волнистой линии на странице книги, но в памяти моего компьютера он представлен 8-битным числовым значением *67*. Это третья буква латинского алфавита, традиционное обозначение шестого химического элемента (углерода) и, так случилось, еще и название языка программирования (§1.6). Все, что имеет значение для программирования строк, это связь между общепринятыми закорючками (называемыми печатными символами) и числовыми кодами. Все еще усложняется тем, что некоторые символы имеют разные числовые коды в разных символьных наборах, не все символы входят во все символьные наборы, и много разных символьных наборов используются на практике. Символьный набор — это соответствие между множеством общеупотребительных символов и их числовых кодов.

Программисты на C++ обычно считают полностью доступным стандартный американский набор символов ASCII, но язык C++ допускает ситуации, когда в среде программирования отсутствуют некоторые из символов. Например, в случае отсутствия символов `[` и `{`, можно использовать ключевые слова и диграфы (digraphs) (§C.3.1).

Расширенные по сравнению с ASCII наборы символов представляют большую проблему. Тексты на таких языках, как, например, китайский, датский, французский, исландский или японский, не могут быть адекватно переданы с помощью лишь ASCII-символов. Хуже того, этим языкам назначены разные, несовместимые между собой символьные наборы. Например, символы европейских языков, применяющих латиницу, почти умещаются в 256-символьный набор. Но, к сожалению, для разных языков по-прежнему используются разные символьные наборы, а разные символы могут иметь один и тот же числовой код. Например, французский (набор Latin-1) плохо уживается с исландским (набор Latin-2). Даже амбициозные проекты по представлению 16-битными кодами всех известных человеку символов (кодировка Unicode) не смогли в полной мере выполнить свою задачу. А 32-битные наборы, способные вместить все что угодно, не нашли широкого применения.

Подход языка C++ состоит в том, чтобы позволить программисту задавать символьный набор в качестве типа символов строки. Можно применить расширенный набор символов или переносимую числовую кодировку (§C.3.3).

20.2.1. Шаблон char_traits

Как показано в §13.2, строки, в принципе, могут использовать любой тип символов, обладающий необходимыми операциями копирования. Однако эффективность работы и простота реализации сильно возрастают для типов со встроенными (а не определяемыми пользователем) операциями копирования. Поэтому стандартный строковый тип ***string*** требует, чтобы типы составляющих строки символов не

имели определяемых пользователем операций копирования. Это также способствует эффективному и простому вводу/выводу строк.

Свойства символьного типа представляются с помощью специализаций шаблона *char_traits*:

```
template<class Ch> struct char_traits { };
```

определенного в пространстве имен *std*. Сам по себе он мало интересен, так как применяются его специализации для конкретных типов символов. Рассмотрим *char_traits<char>*:

```
template<> struct char_trais<char>
{
  typedef char char_type;                                     // тип символа
  static void assign (char_type&, const char_type&) ;         // = для char_type

  // числовое представление символов:
  typedef int int_type;                                       // тип целого для символов

  static char_type to_char_type (const int_type&) ;           // int to char conversion
  static int_type to_int_type (const char_type&) ;            // char to int conversion
  static bool eq_int_type (const int_type&, const int_type&) ; // ==

  // сравнения для char_type:
  static bool eq (const char_type&, const char_type&) ;       // ==
  static bool lt (const char_type&, const char_type&) ;       // <

  // операции над массивами s[n]:
  static char_type* move (char_type* s, const char_type* s2, size_t n) ;
  static char_type* copy (char_type* s, const char_type* s2, size_t n) ;
  static char_type* assign (char_type* s, size_t n, char_type a) ;

  static int compare (const char_type* s, const char_type* s2, size_t n) ;
  static size_t length (const char_type*) ;
  static const char_type* find (const char_type* s, int n, const char_type&) ;

  // все, что связано с I/O:
  typedef streamoff off_type;               // смещение в потоке
  typedef streampos pos_type;               // позиция в потоке
  typedef mbstate_t state_type;             // состояние мультибайтного потока

  static int_type eof () ;                  // end-of-file (конец файла)
  static int_type not_eof (const int_type& i) ; // i, если только i не равно eof()
  static state_type get_state (pos_type p) ;
};
```

Реализация стандартного строкового шаблона, *basic_string* (§20.3), опирается на определенные здесь типы и функции. Тип, применяемый *basic_string* в качестве типа символов, должен поддерживать эти типы и функции.

Чтобы символьный тип был *char_type*, должен иметься способ получения целочисленных значений для каждого символа. Тип этого целого — *int_type*, а преобразования от этого типа к *char_type* и обратно выполняются функциями *to_char_type*() и *to_int_type*(). Для *char* эти преобразования тривиальны. .

Как *move*(*s*, *s2*, *n*), так и *copy*(*s*, *s2*, *n*) используют вызов *assign*(*s*[*i*], *s2*[*i*]), чтобы скопировать *n* символов из *s2* в *s*. Разница состоит в том, что *move*() работает

корректно, даже если [*s,s+n*[и [*s2,s2+n*[перекрываются. Из-за этого *copy*() может оказаться быстрее. Это все напоминает ситуацию с функциями *memcpy*() и *memmove*() из библиотеки языка C (§19.4.6). Вызов *assign*(*s,n,x*) присваивает *n* значений *x* строке *s* при помощи *assign*(*s*[*i*],*x*).

Функция *compare*() использует *lt*() и *eq*() для сравнения символов. Ее возврат имеет тип *int*, причем нуль означает точное совпадение; отрицательное значение свидетельствует о том, что первый аргумент лексикографически предшествует второму, а положительное — что второй предшествует первому. Такой возврат аналогичен возврату функции *strcmp*() (§20.4.1) из библиотеки языка C.

Функции, имеющие отношение к вводу/выводу, используются реализациями низкоуровневых средств ввода/вывода (§21.6.4).

«Широкие» символы — то есть объекты типа *wchar_t* (§4.3), похожи на символы типа *char*, но требуют двух или более байт. Свойства *wchar_t* описываются специализацией *char_traits<wchar_t>*:

```
template<> struct char_traits<wchar_t>
{
  typedef wchar_t char_type;
  typedef wint_t int_type;
  typedef wstreamoff off_type;
  typedef wstreampos pos_type;
  // как char_traits<char>
};
```

Тип *wchar_t* используется для хранения 16-битных символьных наборов, таких как Unicode. При этом *wstreampos* может отличаться от *streampos*, так как может понадобиться хранить состояние смещения (shift state) (§D.4.6).

20.3. Стандартный строковый шаблон basic_string

Строковые средства стандартной библиотеки базируются на шаблоне *basic_string*, предоставляющим типы данных и операции аналогично стандартным библиотечным контейнерам (§16.3):

```
template<class Ch, class Tr = char_traits<Ch>, class A = allocator<Ch> >
class std::basic_string
{
public:
  // ...
};
```

Этот шаблон определен в пространстве имен *std* и расположен в заголовочном файле *<string>*.

С помощью следующих операторов *typedef* задаются общепринятые имена для строковых типов:

```
typedef basic_string<char> string;
typedef basic_string<wchar_t> wstring;
```

Тип *basic_string* похож на *vector* (§16.3), но вместо операций, типичных для списков, он определяет операции, типичные для строк, такие как поиск подстрок. Реа-

лизация ***basic_string*** с помощью ***vector*** или простого массива маловероятна. Типичные строковые операции эффективнее выполняются реализациями, которые минимизируют копирование, не выделяют каждый раз динамическую память под короткие строки, допускают простые модификации длинных строк и т.д. (см. §20.6[12]). Большое количество функций подчеркивает особую роль строкового типа, а также тот факт, что некоторые платформы могут предоставлять аппаратную поддержку строковым операциям. Разработчику библиотеки легче применить аппаратную поддержку, если объявлена стандартная функция со схожей семантикой.

Как и другие стандартные библиотечные типы, ***basic_string<T>*** является конкретным типом (§2.5.3, §10.3) без виртуальных функций. Можно использовать члены такого типа в качестве полей более изощренных классов, предназначенных для реализации специализированных строк, но он не предназначен для использования в качестве базового класса (§25.2.1; см. также §20.6[10]).

20.3.1. Типы

Как и ***vector***, ***basic_string*** определяет связанные с ним типы с помощью оператора ***typedef***:

```
template<class Ch, class Tr = char_traits<Ch>, class A = allocator<Ch> >
class basic_string
{
public:
  // типы (похоже на vector, list и т.д.: §16.3.1):
  typedef Tr traits_type;                    // специфично для basic_string

  typedef typename Tr::char_type value_type;
  typedef A allocator_type;
  typedef typename A::size_type size_type;
  typedef typename A::difference_type difference_type;

  typedef typename A::reference reference;
  typedef typename A::const_reference const_reference;
  typedef typename A::pointer pointer;
  typedef typename A::const_pointer const_pointer;

  typedef implementation_defined iterator;
  typedef implementation_defined const_iterator;

  typedef std::reverse_iterator<iterator> reverse_iterator;
  typedef std::reverse_iterator<const_iterator> const_reverse_iterator;
  // ...
};
```

Класс ***basic_string*** поддерживает строки символов разных типов, а не только строки ***basic_string<char>***, известные как ***string***. Например:

```
typedef basic_string<unsigned char> Ustring;

struct Jchar { /* ... */ };              // тип для японских символов
typedef basic_string<Jchar> Jstring;
```

Строки таких символов могут использоваться точно так же, как строки символов типа ***char***, до тех пор, пока позволяет семантика символьного типа. Например:

```
Ustring first_word(const Ustring& us)
{
  Ustring::size_type pos = us.find(' ');     // см. §20.3.11
  return Ustring(us,0,pos);                  // см. §20.3.4
}

Jstring first_word(const Jstring& js)
{
  Jstring::size_type pos = js.find(' ');     // см. §20.3.11
  return Jstring(js,0,pos);                  // см. §20.3.4
}
```

Естественно, можно использовать шаблоны, принимающие строки в качестве параметров шаблона:

```
template<class S> S first_word(const S& s)
{
  typename S::size_type pos = s.find(' ');  // см. §20.3.11
  return S(s,0,pos);                        // см. §20.3.4
}
```

Объекты типа ***basic_string<Ch>*** могут содержать любые символы из набора ***Ch***. В частности, строки типа ***string*** могут содержать *нуль*. Символьный тип ***Ch*** обязан вести себя так, как ведут себя символы. В частности, он не должен иметь определенных пользователем копирующего конструктора, деструктора и операции присваивания.

20.3.2. Итераторы

Как и другие контейнеры, ***string*** предоставляет итераторы (в том числе и обратные):

```
template<class Ch, class Tr= char_traits<Ch>, class A = allocator<Ch> >
class basic_string
{
public:
  // ...
  // итераторы (как у vector, list и т.д.: §16.3.2):
  iterator begin();
  const_iterator begin() const;
  iterator end();
  const_iterator end() const;

  reverse_iterator rbegin();
  const_reverse_iterator rbegin() const;
  reverse_iterator rend();
  const_reverse_iterator rend() const;
  // ...
};
```

Поскольку ***string*** определяет необходимые типы и имеет функции, возвращающие итераторы, этот тип можно использовать со стандартными алгоритмами (глава 18). Например:

```
void f(string& s)
{
  string::iterator p=find(s.begin(),s.end(),'a');
  //...
}
```

Но наиболее типичные для строк операции класс *string* предоставляет сам. Можно надеяться, что эти методы оптимизируются сильнее, чем это возможно для универсальных алгоритмов.

Стандартные алгоритмы (глава 18) вообще для строк не столь полезны, как можно было бы подумать. Дело в том, что стандартные алгоритмы полагают отдельные элементы контейнеров значимыми в изоляции от других элементов. Для строк же важна именно последовательность следования ее элементов (символов). Поэтому, сортировка универсальных контейнеров улучшает условия их использования, а сортировка строк просто портит их (изменение порядка следования элементов строки лишает ее смысла).

Итераторы для *string* не осуществляют проверку диапазона.

20.3.3. Доступ к элементам (символам)

Доступ к отдельным символам строки возможен с помощью индексирования:

```
template<class Ch, class Tr = char_traits<Ch>, class A = allocator<Ch> >
class basic_string
{
public:
  //...
  // доступ к элементам (как у vector: §16.3.3):
  const_reference operator[](size_type n) const;    // доступ без проверки
  reference operator[](size_type n);

  const_reference at(size_type n) const;             // доступ с проверкой
  reference at(size_type n);
  //...
};
```

При выходе за границы диапазона символов строки функция *at*() генерирует исключение *out_of_range*.

По сравнения с vector, у класса string отсутствуют операции *front*() и *back*(). Чтобы обратиться к первому или последнему символу строки, мы должны написать соответственно *s*[*0*] и *s*[*s.length*()-*1*]. Дуализм «указатель/массив» (§5.3) для типа *string* не имеет места. Если *s* — это строка типа *string*, то &*s*[*0*] — это не то же самое, что *s*.

20.3.4. Конструкторы

Набор операций инициализации и копирования для типа *string* отличается от такового для остальных контейнеров (§16.3.4) во многих деталях:

```
template<class Ch, class Tr = char_traits<Ch>, class A = allocator<Ch> >
class basic_string
{
public:
```

```
    // ...
    // конструкторы и т.д. (немного похоже на vector и list: §16.3.4):
    explicit basic_string (const A& a = A () ) ;
    basic_string (const basic_string& s, size_type pos = 0,
            size_type n = npos, const A& a = A () ) ;
    basic_string (const Ch* p, size_type n, const A& a = A () ) ;
    basic_string (const Ch* p, const A& a = A () ) ;
    basic_string (size_type n, Ch c, const A& a = A () ) ;
    template<class In> basic_string (In first, In last, const A& a = A () ) ;

    ~basic_string () ;

    static const size_type npos;
    // ...
};
```

Строку типа *string* можно инициализировать C-строкой, другой строкой типа *string*, подстроками C-строк или строк типа *string*, или последовательностью символов. Но ее нельзя инициализировать символом или числом:

```
void f (char* p, vector<char>& v)
{
    string s0;                          // пустая строка
    string s00 = "" ;                   // тоже пустая строка

    string s1 = 'a' ;                   // error: нет приведения из char в string
    string s2 = 7;                      // error: нет приведения из int в string
    string s3 (7) ;                     // error: нет соответствующего конструктора

    string s4 (7, 'a' ) ;               // 7 копий 'a', то есть "aaaaaaa"

    string s5 = "Frodo" ;               // копия "Frodo"
    string s6 = s5;                     // копия s5

    string s7 (s5, 3, 2) ;              // s5[3] и s5[4]; то есть "do"
    string s8 (p+7, 3) ;                // p[7],p[8] и p[9]
    string s9 (p, 7, 3) ;               // string(string(p),7,3), расточительно

    string s10{v.begin () ,v.end () } ; // копирует все символы из v
}
```

Символы нумеруются начиная с нуля, так что строка — это последовательность символом с номерами от *0* до *length*() *-1*.

Функция *length*() есть фактически синоним для *size*(): обе функции возвращают количество символов в строке. Отметим, что работа этих функций не базируется на понятии терминального нуля, заканчивающего собой C-строки (§20.4.1). Реализации *basic_string* хранят длину строк, не полагаясь на терминальный нуль.

Подстроки фиксируются начальной позицией и числом символов. По умолчанию npos инициализируется максимально возможным числом символов, трактуемым как «все элементы».

Не существует конструктора, который создает строку из n неопределенных символов. Ближайшим по смыслу является конструктор, который создает строку из n копий заданного символа. Отсутствие конструкторов, которые принимают единственный символ или одно лишь число символов, позволяет компилятору обнаруживать ошибки, такие как определение *s1*, *s2* и *s3* в вышеприведенном примере.

Копирующий конструктор принимает четыре аргумента. Три из них имеют значения по умолчанию. Для эффективности такой конструктор можно реализовать в виде двух отдельных конструкторов. Пользователь может догадаться об этом, лишь посмотрев на сгенерированный код.

Шаблонный конструктор классового шаблона ***basic_string*** является наиболее универсальным. Он позволяет инициализировать строку значениями из произвольной последовательности. В частности, он может инициализировать строку элементами различных символьных типов, для которых имеются необходимые преобразования. Например:

```
void f(string s)
{
  wstring ws(s.begin(),s.end());    // копируются все символы из s
  // ...
}
```

Каждый элемент типа ***wchar_t*** в строке ***ws*** инициализируется соответствующим элементом типа ***char*** из ***s***.

20.3.5. Ошибки

Когда строки просто считывают, записывают, печатают, хранят, сравнивают, копируют и так далее, то при этом не возникает никаких проблем, кроме, возможно, производительности. Иное дело, когда мы начинаем использовать отдельные символы или подстроки для того, чтобы из существующих строк получить новые строки — тут легко нарваться на ошибки, связанные с попытками записи за конец строки.

При явных обращениях к отдельным символам строки функцией ***at***(), она генерирует исключение ***out_of_range*** в случае выхода за границу диапазона строковых элементов; операция [] не делает этого.

Многие строковые операции на входе принимают позицию символа и число элементов. Если заданная позиция символа превышает длину строки, генерируется исключение ***out_of_range***. Если же заданное число элементов «слишком велико», то оно просто трактуется как «все остальные символы». Например:

```
void f()
{
  string s = "Snobol4";
  string s2(s,100,2);              // символ за концом строки: генерируется out_of_range
  string s3(s,2,100);              // слишком много символов: эквив-но s3(s,2,s.size()-2)
  string s4(s,2,string::npos);     // символы, начиная с s[2]
}
```

Таким образом, «слишком больших» позиций нужно избегать, в то время как «слишком много» символов могут оказаться полезными. Фактически, ***npos*** есть наибольшее возможное для типа ***size_type*** число.

Можно попробовать задать отрицательные значения для позиции и числа элементов:

```
void g (string& s)
{
  string s5 (s, -2, 3) ;        // большая позиция!: генерируется out_of_range
  string s6 (s, 3, -2) ;        // большое число символов!: все ok
}
```

Но поскольку ***size_type***, применяемый и для позиции, и для числа элементов, есть тип беззнаковый, то отрицательные числа служат лишь запутанным способом задания больших положительных значений (§16.3.4).

Обратите внимание на то, что функции, призванные находить подстроки в строках типа ***string*** (§20.3.11), в случае неудачи возвращают ***npos***. То есть они не возбуждают исключений. Однако при последующем использовании ***npos*** в качестве исходной позиции символа исключение генерируется.

Другой способ задания подстроки заключается в указании пары итераторов. Первый итератор задает начальный символ, а разность итераторов указывает число элементов. Как обычно, итераторы не проверяют выход за границу диапазона элементов.

В случае применения С-строк проверка диапазона символов лишь только усложняется. Когда функции класса ***basic_string*** принимают С-строку (указатель на ***char***), они полагают указатель не равным нулю. Принимая позицию символа в С-строке, они предполагают, что строка достаточно длинна и позиция символа находится в ее пределах. Будьте осторожны! Здесь требуется прямо-таки параноидальная осторожность (кроме как в случае литералов).

Для всех строк ***length()<npos***. В редких случаях, таких как вставка одних строк в другие (§20.3.9), может образовываться слишком длинная строка, которую невозможно представить, и тогда генерируется исключение ***length_error***. Например:

```
string s (string::npos, 'a') ;        // генерируется length_error()
```

20.3.6. Присваивание

Естественно, что для строк определены присваивания:

```
template<class Ch, class Tr = char_traits<Ch>, class A = allocator<Ch> >
class basic_string
{
public:
  // ...
  // присваивание (немного похоже на vector и list: §163.4):
  basic_string& operator= (const basic_string& s) ;
  basic_string& operator= (const Ch* p) ;
  basic_strings operator= (Ch c) ;

  basic_string& assign (const basic_string& s) ;
  basic_string& assign (const basic_string& s, size_type pos, size_type n) ;
  basic_string& assign (const Ch* p, size_type n) ;
  basic_string& assign (const Ch* p) ;
  basic_string& assign (size_type n, Ch c) ;
  template<class In> basic_string& assign (In first, In last) ;
  // ...
};
```

Как и другие стандартные контейнеры, строки типа ***string*** имеют семантику значений. То есть когда строка присваивается другой строке, ее содержимое копируется и после присваивания существуют две отдельные строки с одинаковым значением. Например:

```
void g()
{
  string s1 = "Knold";
  string s2 = "Tot";

  s1 = s2;           // две копии "Tot"
  s2[1] = 'u';       // s2 есть "Tut", s1 по-прежнему "Tot"
}
```

Разрешено присваивание строке одиночного символа, хотя такой инициализации не существует:

```
void f()
{
  string s = 'a';    // error: инициализация типом char
  s = 'a';           // ok: присваивание
  s = "a";
  s = s;
}
```

Присваивание одиночных значений типа ***char*** строкам типа ***string*** не столь полезно и чревато ошибками. В то же время, операция += для одиночных символов бывает полезна (§20.3.9), а допустить лишь *s*+='*c*' без допуска *s*=*s*+'*c*' было бы странным.

Функции ***assign***() используются для присваиваний, сопоставляемым конструкторам со многими аргументами (§16.3.4, §20.3.4).

Как упоминалось в §11.12, тип ***string*** можно оптимизировать так, чтобы копирование фактически выполнялось лишь тогда, когда на самом деле требуются две раздельные копии строки. Класс ***string*** запроектирован таким образом, что он поощряет создание реализаций, минимизирующих фактическое копирование. Это делает применение строк «только для чтения» и их передачу в качестве аргументов функциям более дешевым. Но программистам следует детально ознакомиться с конкретными реализациями, прежде чем полагаться на такую оптимизацию (§20.6[13]).

20.3.7. Преобразование в С-строку

Как показано в §20.3.4, строки типа ***string*** можно инициализировать С-строками, а также этим строкам можно присваивать С-строки. И наоборот, можно в С-строку поместить копии символов строки типа ***string***:

```
template<class Ch, class Tr = char_traits<Ch>, class A = allocator<Ch> >
class basic_string
{
public:
  // ...
  // преобразование в С-строку:
  const Ch* c_str() const;
```

```
    const Ch* data() const;
    size_type copy(Ch* p, size_type n, size_type pos = 0) const;
    // ...
};
```

Функция *data*() пишет символы строки в массив и возвращает указатель на этот массив. Массив принадлежит строке типа *string* и пользователь не должен пытаться его уничтожать. Пользователь также не должен полагаться на содержимое массива после выполнения над строкой неконстантной операции (вызова *неконстантной* функции). Функция *c_str*() похожа на *data*(), но добавляет в конец массива терминальный нуль. Например:

```
void f()
{
    string s = "equinox";            // s.length()== 7
    const char* p1 = s.data();       // p1 указывает на 7 символов
    printf("p1 = %s\n",p1);          // плохо: отсутствует терминальный нуль

    p1[2]  = 'a';                    // error: p1 указывает на константный массив
    s[2]   = 'a';
    char c = p1[1];                  // плохо: доступ к s.data() после модификации s

    const char* p2 = s.c_str();      // p2 указывает на восемь символов
    printf("p2 = %s\n" ,p2);         // ok: c_str() добавляет терминальный нуль
}
```

Другими словами, *data*() создает массив символов, в то время как *c_str*() создает C-строку. Эти функции предназначены в первую очередь для того, чтобы упростить применение функций, использующих C-строки. Следовательно, *c_str*() представляется более полезной, чем *data*(). Например:

```
void f(string s)
{
    int i = atoi(s.c_str());         // получает числовое значение из цифр строки (§20.4.1)
    // ...
}
```

В типичном случае, пока вам символы непосредственным образом не требуются, лучше оставить их в составе строки типа *string*. В то же время, если вы не можете сразу распорядиться ими, то символы можно скопировать и во внешний буфер, а не оставлять их во временных буферах, выделяемых функциями *c_str*() или *data*(). Для этого предназначена функция *copy*(). Например:

```
char* c_string(const string& s)
{
    char* p = new char[s.length()+1];   // внимание: +1 для терминального нуля
    p[s.copy(p,string::npos)] = 0;      // внимание: добавляем терминатор и return p;
    return p;
}
```

Вызовом *s.copy*(*p*,*n*,*m*) в область памяти, указуемую через *p*, копируется не более *n* символов, начиная с *s*[*m*]. Если в строке *s* символов меньше, чем *n*, функция *copy*() просто копирует все имеющиеся символы.

Отметим, что строки типа *string* могут содержать нули. Функции, манипулирующие С-строками, воспримут эти нули в качестве терминальных. Поэтому вносите нули либо если вы не собираетесь обращаться к функциям, работающим с С-строками, либо вносите их именно в качестве терминальных.

Для преобразования к С-строкам предназначена также и операция ***operator const char**** (), а не только функция ***c_str*** (). Она дает удобство неявного приведения вместе с сюрпризами в виде непреднамеренных преобразований.

Если функция ***c_str*** () слишком часто вызывается в вашей программе, то скорее всего вы излишне опираетесь на интерфейсы в С-стиле. Почти всегда есть возможность полностью перейти на интерфейсы класса *string*, что устранит необходимость в лишних преобразованиях. Можно и по-другому избавиться от явных вызовов функции ***c_str*** (), просто определив свои собственные функции вместо тех, которые требуют вызовов ***c_str*** ():

```
extern C" int atoi (const char*) ;

int atoi (const string& s)
{
  return atoi (s.c_str ()) ;
}
```

20.3.8. Сравнения

Строки можно сравнивать со строками их же типа и с массивами элементов того же самого символьного типа:

```
template<class Ch, class Tr = char_traits<Ch>, class A = allocator<Ch> >
class basic_string
{
public:
  // ...
  int compare (const basic_string& s) const;          // комбинируются > и ==
  int compare (const Ch* p) const;

  int compare (size_type pos, size_type n, const basic_string& s) const;
  int compare (size_type pos, size_type n, const basic_string& s,
               size_type pos2, size_type n2) const;
  int compare (size_type pos, size_type n, const Ch* p, size_type n2 = npos) const;
  // ...
};
```

Когда строке передаются позиция и размер в виде параметров функции ***compare*** (), используется лишь обозначенная тем самым подстрока. Например, ***s.compare (pos, n, s2)*** эквивалентно ***string (s, pos, n) .compare (s2)***. В качестве критерия сравнения используется ***compare*** () из ***char_traits<Ch>*** (§20.2.1). Таким образом, ***s.compare (s2)*** возвращает ***нуль***, если строки имеют одинаковое значение; отрицательное число, если ***s*** лексикографически предшествует ***s2***, и положительное значение в противном случае.

Пользователь не может задать свой критерий сравнения, как это делалось в §13.4. Если нужен такой уровень гибкости, мы можем применить ***lexicographical_compare*** () (§18.9), определить функцию, как в §13.4, или написать явный цикл. Например, функция ***toupper*** () (§20.4.2) позволит нам выполнить сравнение без учета регистра символов:

```
int cmp_nocase(const string& s, const string& s2)
{
  string::const_iterator p = s.begin();
  string::const_iterator p2 = s2.begin();

  while(p!=s.end() && p2!=s2.end())
  {
    if(toupper(*p) !=toupper(*p2))
      return (toupper(*p)<toupper(*p2)) ? -1 : 1;
    ++p;
    ++p2;
  }

  return (s2.size()==s.size()) ? 0 : (s.size()<s2.size()) ? -1 : 1;
}

void f(const string& s, const string& s2)
{
  if(s == s2)                    // сравнение s и s2 с учетом регистра
  {
    // ...
  }

  if(cmp_nocase(s,s2) == 0)      // сравнение s и s2 без учета регистра
  {
    // ...
  }
  // ...
}
```

Для типа ***basic_string*** определены обычные операции сравнения ==, !=, >, <, >= и <=:

```
template<class Ch, class Tr, class A>
bool operator==(cons basic_string<Ch,Tr,A>&, const basic_string<Ch,Tr,A>&);

template<class Ch, class Tr, class A>
bool operator==(const Ch*,const basic_string<Ch,Tr,A>&);

template<class Ch, class Tr, class A>
bool operator==(const basic_string<Ch,Tr,A>&, const Ch*);

// аналогичные объявления для !=,>, <,>= и <=
```

Операции сравнения определены вне класса ***basic_string*** (то есть не являются его членами), так что преобразование типов одинаковым образом применимо к обоим операндам (§11.2.3). Версии, работающие с С-строками, введены для оптимизации сравнения с литералами. Например:

```
void f(const string& name)
{
  if(name =="Obelix" || "Asterix" ==name)   // используется оптимизированная ==
  {
    // ...
  }
}
```

20.3.9. Вставка

Когда строка создана, ею можно манипулировать разными способами. Из всех операций, изменяющих содержимое строки, наиболее распространенной является *операция добавления новых символов в конец строки* (*appending*). Вставка символов в иные позиции внутри строки встречается реже:

```
template<class Ch, class Tr = char_traits<Ch>, class A = allocator<Ch> >
class basic_string
{
public:
  // ...
  // добавление символов после (*this)[length()-l];
  basic_string& operator+= (const basic_string& s) ;
  basic_string& operalor+= (const Ch* p) ;
  basic_string& operator+= (Ch c) ;
  void push_back (Ch c) ;

  basic_string& append (const basic_string& s) ;
  basic_string& append (const basic_string& s, size_type pos, size_type n) ;
  basic_string& append (const Ch* p, size_type n) ;
  basic_string& append (const Ch* p) ;
  basic_string& append (size_type n, Ch c) ;
  template<class In> basic_string& append (In first, In last) ;

  // вставка символов перед (*this)[pos]: ,
  basic_string& insert (size_type pos, const basic_string& s) ;
  basic_string& insert (size_type pos, const basic_string& s, size_type pos2, size_type n) ;
  basic_string& insert (size_type pos, const Ch* p, size_type n) ;
  basic_string& insert (size_type pos, const Ch* p) ;
  basic_string& insert (size_type pos, size_type n, Ch c) ;

  // вставка символов перед p:
  iterator insert (iterator p, Ch c) ;
  void insert{ iterator p, size_type n, Ch c) ;
  template<class In> void insert (iterator p, In first, In last) ;
  // ...
};
```

Операция += обеспечивает наглядное отображение для большинства операций добавления символов. Например:

```
string complete_name (const string& first_name, const string& family_name)
{
  string s = first_name;
  s += ' ';
  s += family_name;
  return s;
}
```

Добавление символов в конец строки часто намного эффективнее вставок символов в иные позиции. Например:

```
string complete_name2 (const string& first_name, const string& family_name)
// плохой алгоритм
```

```
{
  string s = family_name;
  s.insert(s.begin(), ' ');
  return s.insert(0,first_name);
}
```

В общем случае вставки вынуждают реализацию **string** выделять дополнительную память и переписывать символы с места на место.

Поскольку **string** имеет операцию **push_back**() (§16.3.5), для строк можно применять **back_inserter**() так же, как для любых стандартных контейнеров.

20.3.10. Конкатенация

Добавление символов в конец — это специальный случай конкатенации. *Конкатенация* (*concatenation*) — формирование строки из двух строк их сцеплением, выполняется операцией +:

```
template<class Ch, class Tr, class A>
basic_string<Ch,Tr,A>
operator+(const basic_string<Ch,Tr,A>&, const basic_string<Ch,Tr,A>&);

template<class Ch, class Tr, class A>
basic_string<Ch Tr,A> operator+(const Ch*, const basic_string<Ch,Tr,A>&);

template<class Ch, class Tr, class A>
basic_string<Ch,Tr,A> operator+(Ch, const basic_string<Ch,Tr,A>&);

template<class Ch, class Tr, class A>
basic_string<Ch,Tr,A> operator+(const basic_string<Ch,Tr,A>&, const Ch*);

template<class Ch, class Tr, class A>
basic_string<Ch,Tr,A> operator+(const basic_string<Ch,Tr,A>&, Ch);
```

Как обычно, операция + определяется вне класса **basic_string** (не является его членом). Для шаблонов со многими параметрами это приводит к некоторой тяжеловесности записи в связи с многократными упоминаниями параметров.

С другой стороны, применение конкатенации удобно и наглядно. Например:

```
string complete_name3(const string& first_name, const string& family_name)
{
  return first_name + ' ' + family_name;
}
```

Удобство записи достигается за счет некоторого понижения эффективности по сравнению с функцией **complete_name**(), так как **complete_name3**() нуждается в создании одной дополнительной временной переменной (§10.4.10). Мой опыт показывает, что это редко когда имеет большое значение, но об этом все же лучше помнить при написании вложенных циклов. В наиболее критичных ситуациях можно организовать функцию **complete_name**() как встраиваемую или выполнить композицию строк с помощью низкоуровневых операций (§20.6[14]).

20.3.11. Поиск

Имеется огромное количество функций поиска:

```
template<class Ch, class Tr = char_traits<Ch>, class A = allocator<Ch> >
class basic_string
{
public:
  // ...
  // поиск подпоследовательности (вроде search() §18.5.5):
  size_type find (const basic_string& s, size_type i = 0) const;
  size_type find (const Ch* p, size_type i, size_type n) const;
  size_type find (const Ch* p, size_type i = 0) const;
  size_type find (Ch c, size_type i = 0) const;

  // обратный поиск подпоследовательности с конца (вроде find_end(), §18.5.5):
  size_type rfind (const basic_string& s, size_type i = npos) const;
  size_type rfind (const Ch* p, size_type i, size_type n) const;
  size_type rfind (const Ch* p, size_type i = npos) const;
  size_type rfind (Ch c, size_typei = npos) const;

  // поиск первого символа (любого), входящего в аргумент:
  size_type find_first_of(const basic_string& s, size_type i = 0) const;
  size_type find_first_of(const Ch* p, size_type i, size_type n) const;
  size_type find_first_of(const Ch* p, size_type i = 0) const;
  size_type find_first_of(Ch c, size_type i = 0) const;

  // поиск последнего символа (любого), входящего в аргумент:
  size_type find_last_of(const basic_string& s, size_type i = npos) const;
  size_type find_last_of(const Ch* p, size_type i, size_type n) const;
  size_type find_last_of(const Ch* p, size_type i = npos) const;
  size_type find_last_of(Ch c, size_type i = npos) const;

  // поиск первого символа, не входящего в аргумент:
  size_type find_first_not_of(const basic_string& s, size_type i = 0) const;
  size_type find_first_not_of(const Ch* p, size_type i, size_type n) const;
  size_type find_first_not_of(const Ch* p, size_type i = 0) const;
  size_type find_first_not_of(Ch c, size_type i = 0) const;

  // поиск последнего символа, не входящего в аргумент:
  size_type find_last_not_of(const basic_string& s, size_type i = npos) const,
  size_type find_last_not_of(const Ch* p, size_type i, size_type n) const;
  size_type find_last_not_of(const Ch*p, size_type i = npos) const;
  size_type find_last_not_of(Ch c, size_type i = npos) const;
  // ...
};
```

Все это константные функции-члены. Они предназначены для локализации искомой строки, но с их помощью нельзя изменить исходную строку.

Смысл и назначение функций *basic_string::find()* можно понять, сравнив их с аналогичными стандартными алгоритмами. Рассмотрим пример:

```
void f()
{
  string s = "accdcde";
```

```
    string::size_type i1 = s.find("cd");              // i1 = 2  s[2]=='c'&&s[3]=='d'
    string::size_type i2 = s.rfind("cd");             // i2 = 4  s[4]=='c'&&s[5]=='d'
    string::size_type i3 = s.find_first_of("cd");     // i3 = 1  s[1]=='c'
    string::size_type i4 = s.find_last_of("cd");      // i4 = 5  s[5]=='d'
    string::size_type i5 = s.find_first_not_of("cd"); // i5 = 0  s[0]!='c'&&s[0]!='d'
    string::size_type i6 = s.find_last_not_of("cd");  // i6 = 6  s[6]!='c'&&s[6]!='d'
}
```

Если функция *find*() ничего не находит, она возвращает *npos*, что означает недопустимую позицию символа. Применение npos в качестве позиции символа вызывает генерацию исключения *out_of_range* (§20.3.5).

Отметим, что возврат *find*() есть число беззнаковое.

20.3.12. Замена

После того как позиция символа в строке определена, значения индивидуальных символов можно изменить при помощи операции индексирования, или можно заменить целую подстроку новыми символами при помощи функции *replace*():

```
template<class Ch, class Tr = char_traits<Ch>, class A = allocator<Ch> >
class basic_string
{
public:
  // ...
  // замена [(*this)[i],(*this)[i+n][ другими символами:
  basic_string& replace(size_type i, size_type n, const basic_string& s);
  basic_string& replace(size_type i, size_type n,
                        const basic_string& s, size_type i2, size_type n2);
  basic_string& replace(size_type i, size_type n, const Ch* p, size_type n2);
  basic_string& replace(size_type i, size_type n, const Ch* p);
  basic_string& replace(size_type i, size_type n, size_type n2, Ch c);

  basic_string& replace(iterator i, iterator i2, const basic_string& s);
  basic_string& replace(iterator i, iterator i2, const Ch* p, size_type n);
  basic_string& replace(iterator i, iterator i2, const Ch* p);
  basic_string& replace(iterator i, iterator 12, size_type n, Ch c);
  template<class In> basic_string& replace(iterator i, iterator i2, In j, In j2);

  // удаление символов из строки ("замена ничем"):
  basic_string& erase(size_type i = 0, size_type n = npos);
  iterator erase(iterator i);
  iterator erase(iterator first, iterator last);
  void clear();                                     // удаляет все символы
  // ...
};
```

Отметим, что число новых символов может не совпадать с числом символов подстроки. Размер строки изменяется так, чтобы соответствовать новому содержимому подстроки. В частности, *erase*() просто удаляет подстроку и соответственно изменяет размер строки. Например:

```
void f()
{
  string s = "but I have heard it works even if you don' t believe in it";
  s.erase(0,4);                              // удаляем начальное "but "
  s.replace(s.find("even"),4,"only");
  s.replace(s.find("don't"),5,"");           // удаляем с заменой на ""
}
```

Как и вызов ***clear***(), вызов ***erase***() без аргументов просто превращает любую строку в пустую строку.

Набор функций ***replace***() соответствует таковому для функций присваивания. В конце концов, ***replace***() — это присваивание нового значения подстроке.

20.3.13. Подстроки

Функция ***substr***() позволяет задать подстроку, указав начальную позицию и длину:

```
template<class Ch, class Tr = char_traits<Ch>, class A = allocator<Ch> >
class basic_string
{
public:
  // ...
  // адрес подстроки:
  basic_string substr(size_type i = 0, size_type n = npos) const;
  // ...
};
```

Функция ***substr***() — это способ прочитать часть строки. С другой стороны, функция ***replace***() позволяет выполнять запись в подстроку. Обе функции нуждаются в задании позиции (низкоуровневой) символа плюс число символов. В то же время ***find***() позволяет находить подстроки по их значениям. Вместе они позволяют нам находить подстроки, которые можно применять как для чтения, так и для записи:

```
template<class Ch> class Basic_substring
{
public:
  typedef typename basic_string<Ch>::size_type size_type;

  Basic_substring(basic_string<Ch>& s, size_type i, size_type n);   // s[i]..s[i+n-l]
  Basic_substring(basic_string<Ch>& s, const basic_string<Ch>& s2); // s2 в s
  Basic_substring(basic_string<Ch>& s, const Ch* p);                // *p в s

  Basic_substring operator=(const basic_string<Ch>&);
  Basic_substring operator=(const Basic_substring<Ch>&);
  Basic_substring operator=(const Ch*);
  Basic_substring operator=(Ch);

  operator basic_string<Ch>() const;
  operator const Ch*() const;

private:
  basic_string<Ch>* ps;
```

```
    size_type pos;
    size_type n;
};
```

Реализация весьма тривиальна. Например:

```
template<class Ch>
Basic_substring<Ch>::Basic_substring(basic_string<Ch>& s, const basic_string<Ch>& s2)
        : ps(&s), n(s2.length())
{
    pos = s.find(s2);
}

template<class Ch>
Basic_substring<Ch>& Basic_substring<Ch>::operator=(const basic_string<Ch>& s)
{
    ps->replace(pos, n, s);
    return *this;
}

template<class Ch> Basic_substring<Ch>::operator basic_string<Ch>() const
{
    return basic_string<Ch>(*ps, pos, n);
}
```

Если *s2* не находится внутри *s*, то значение *pos* равно *npos*. Чтение или запись с таким значением генерирует исключение *out_of_range* (§20.3.5).

Применить *Basic_substring* можно следующим образом:

```
typedef Basic_substring<char> Substring;

void f()
{
    string s = "Mary had a little lamb";
    Substring(s, "lamb") = "fun";
    Substring(s, "a little") = "no";
    string s2 = "Joe" + Substring(s, s.find(' '), string::npos);
}
```

Естественно, было бы еще интереснее, если бы *Substring* искал бы подстроки по шаблонному образцу (§20.6[7]).

20.3.14. Размер и емкость

Проблемы с памятью решаются в том же ключе, что и у класса *vector* (§16.3.8):

```
template<class Ch, class Tr = char_traits<Ch>, class A = allocator<Ch> >
class basic_string
{
public:
    // ...
    // размер, емкость и пр. (подобно §16.3.8):
    size_type size() const;                 // число символов (§20.3.4)
    size_type max_size() const;             // максимальная длина строки
    size_type length() const {return size();}
    bool empty() const {return size()==0;}
```

```
    void resize (size_type n, Ch c) ;
    void resize (size_type n) {resize (n, Ch ( ) ) ; }

    size_type capacity ( ) const;                    // как у vector: §16.3.8
    void reserve (size_type res_arg = 0) ;           // как у vector: §16.3.8

    allocator_type get_allocator ( ) const;
};
```

Вызов ***reserve***(***res_arg***) генерирует исключение ***length_error***, если ***res_arg>max_size***().

20.3.15. Операции ввода/вывода

Одно из важнейших применений типа ***string*** заключается в использовании строк этого типа для приема ввода, или в качества источника вывода. Операции ввода/вывода для ***basic_string*** представлены в заголовочном файле <***string***> (а не в <***iostream***>):

```
template<class Ch, class Tr, class A>
basic_istream<Ch, Tr>& operator>> (basic_istream<Ch, Tr>&, basic_string<Ch, Tr, A>&) ;

template<class Ch, class Tr, class A>
basic_ostream<Ch, Tr>& operator<< (basic_ostream<Ch, Tr>&, const
basic_string<Ch, Tr, A>&) ;

template<class Ch, class Tr, class A>
basic_istream<Ch, Tr>& getline (basic_istream<Ch, Tr>&, basic_string<Ch, Tr, A>&, Ch eol) ;

template<class Ch, class Tr, class A>
basic_istream<Ch, Tr>& getline (basic_istream<Ch, Tr>&, basic_string<Ch, Tr, A>&) ;
```

Операция << пишет строку в поток ***ostream*** (§21.2.1). Операция >> читает слово, ограниченное пробельным символом (whitespace-terminated word) (§3.6, §21.3.1), в строку, расширяя ее по мере необходимости (чтобы уместилось все слово). Начальные пробельные символы пропускаются, и терминальный пробельный символ также в строку не попадает.

Функция ***getline***() читает строчку текста вплоть до символа ***eol***, расширяя приемную строку по мере необходимости, чтобы вместить весь прочитанный текст (§3.6). Если аргумент ***eol*** не указывается, в качестве ограничителя используется символ '\n'. Ограничитель удаляется из потока ввода, но в приемную строку не попадает. Поскольку приемная строка автоматически расширяется, нет необходимости оставлять ограничитель в потоке или подсчитывать число прочитанных символов так, как это делают функции ***get***() или ***getline***() для символьных массивов (§21.3.4).

20.3.16. Обмен строк

Как и для векторов (§16.3.9), для типа ***string*** функция ***swap***() может оказаться эффективнее универсального алгоритма:

```
template<class Ch, class Tr, class A>
void swap (basic_string<Ch, Tr, A>&, basic_string<Ch, Tr, A>&) ;
```

20.4. Стандартная библиотека языка С

Стандартная библиотека языка С++ унаследовала функции для манипуляции С-строками из стандартной библиотеки языка С. В настоящий раздел включены лишь самые полезные из них. Описание не претендует на полноту; за дальнейшей информацией обращайтесь к справочным руководствам. Имейте ввиду, что разработчики конкретных реализаций часто добавляют в стандартные заголовочные файлы свои нестандартные функции, так что непросто бывает выяснить, какие функции доступны для всех реализаций, а какие нет.

Заголовочные файлы, предоставляющие возможности стандартной библиотеки языка С, перечислены в §16.1.2. Функции для работы с памятью рассматриваются в §19.4.6, функции ввода/вывода — в §21.8, а математические функции — в §22.3. Функции, имеющие отношение к запуску программ и их завершению, рассматриваются в §3.2 и §9.4.1.1, а средства для работы с неуказанным числом функциональных параметров — в §7.6. Функции для работы с «широкими» строками можно найти в *<cwchar>* и *<wchar.h>*.

20.4.1. С-строки

Функции для манипуляции С-строками представлены в заголовочных файлах *<string.h>* и *<cstring>*:

```
char* strcpy(char* p, const char* q);         // копирование из q в p (вкл. терминатор)
char* strcat(char* p, const char* q);         // добавление q к p (вкл. терминатор)
char* strncpy(char* p, const char* q, int n); // копирование n char из q в p
char* strncat(char* p, const char* q, int n); // добавление n char из q в p

size_t strlen(const char* p);                 // длина p (без учета терминатора)

int strcmp(const char* p, const char* q);     // сравниваем p и q
int strncmp(const char* p, const char* q, int n); // сравниваем первые n char

char* strchr(char* p, int c);                 // ищем первое c в p
const char* strchr(const char* p, int c);
char* strrchr(char* p, int c);                // ищем последнее c в p
const char* strrchr(const char* p, int c);
char* strstr(char* p, const char* q);         // ищем первое q в p
const char* strstr(const char* p, const char* q);

char* strpbrk(char* p, const char* q);        // ищем первый char из q в p
const char* strpbrk(const char* p, const char* q);

size_t strspn(const char* p, const char* q);  // число char в p до любого char не в q
strcspn(const char* p,const char* q);         // число char в p до любого char из q
```

Предполагается, что указатель не равен нулю, а массив элементов типа ***char***, на который он указывает, завершается *нулем*. Функции, в именах которых присутствует ***strn***, в случае недобора символов до числа ***n***, заполняют вакансии *нулями*. При сравнении одинаковых строк возвращается *нуль*; отрицательное число возвращается, если первая строка лексикографически предшествует второй, и положительное число в противном случае.

Естественно, в С нельзя определить функции с одинаковыми именами (перегрузка функций). Но в С++ они нужны для предоставления безопасных ***const***-версий. Например:

```
void f(const char* pcc, char* pc)  // C++
{
  *strchr(pcc,'a') = 'b' ;        // error: нельзя присвоить const char
  *strchr(pc,'a')  = 'b';         // ok но не аккуратно: в pc может не быть 'a'
}
```

Функцию ***strchr***() в С++ невозможно использовать для записи в *константную* строку. В то же время С-программа может проэксплуатировать более слабую проверку типов:

```
char* strchr(const char* p, int c) ;  /* библиотечная фун-я языка С, а не С++ */

void g(const char* pcc, char* pc)     /* С, а в С++ не скомпилируется */
{
  strchr(pcc,'a') = 'b' ;      /* приводит const к non-const: ok в С, error в С++ */
  strchr(pc,'a')  = 'b';       /* ok на С и С++ */
}
```

Всюду, где только возможно, следует использовать тип ***string*** вместо С-строк. Конечно, С-строки и предназначенные для их обработки стандартные функции библиотеки языка С могут обеспечить высокую эффективность, но даже опытные программисты при работе с ними допускают неочевидные («глупые») ошибки. В то же время С++-программисты неизбежно с ними сталкиваются, просматривая старый С-код. Вот соответствующий типичный пример:

```
void f(char* p, char* q)
{
  if(p==q) return;          // указатели равны
  if(strcmp(p,q) ==0)       // значения строк равны
  {
    int i = strlen(p) ;     // число символов (без учета терминатора)
    // . . .
  }

  char buf[200] ;
  strcpy(buf,p) ;           // копируем p в buf (включая терминатор)
                            // опасно: когда-нибудь переполнится.
  strncpy(buf,p,200) ;      // копируем 200 char из p в buf
                            // опасно: когда-нибудь не скопируется терминатор.
}
```

Ввод и вывод С-строк обычно выполняется с помощью функций семейства ***printf***() (§21.8).

В заголовочных файлах <***stdlib.h***> и <***cstdlib***> стандартная библиотека объявляет полезные функции для конвертации строк, представляющих числовые значения, в числа. Например:

```
double atof(const char* p) ;                // преобразует p[] в double
double strtod(const char* p, char** end) ;  // преобразует p[] в double
int atoi(const char* p) ;                   // преобразует p[] в int (основание 10)
```

```
long atol(const char* p);                      // преобразует p[] в long (основание 10)
long strtol(const char* p, char** end, int b); // преобразует p[] в long (основание b)
```

Пробельные символы в начале строк игнорируются. Если строка не представляет собой числового значения, возвращается нуль. Например, ***atoi("seven")*** дает ***0***.

Если в вызове ***strtol(p,end,b)*** ***end*** не равно *нулю*, непрочитанная часть строки доступна с помощью ****end***. Если ***b==0***, то число интерпретируется как целый литерал C++ (§4.4.1); например, префикс ***0x*** соответствует шестнадцатеричной форме записи числа, ***0*** — восьмеричной и т.д.

Возвращаемые функциями ***atof()***, ***atoi()*** и ***atol()*** числа не определены тогда, когда они пытаются преобразовать число, не отображаемое с помощью типов возврата этих функций. Если входная строка функции ***strtol()*** содержит текстовое представление числа, которое не может быть отображено в типе ***long int***, или в вызове функции ***strtod()*** строка содержит представление числа, неотобразимого в ***double***, то ***errno*** принимает значение ***ERANGE***, и возвращается очень большое (или очень маленькое) значение.

За исключением обработки ошибок, ***atof(s)*** эквивалентна ***strtod(s,0)***, ***atoi(s)*** эквивалентна ***int(strtol(s, 0, 10))***, и ***atol(s)*** эквивалентна ***strtol(s, 0, 10)***.

20.4.2. Классификация символов

В заголовочных файлах <***ctype.h***> и <***cctype***> стандартная библиотека объявляет полезные функции для работы с ASCII-символами и другими символьными наборами:

```
int isalpha(int);   // буквы: 'a'..'z' 'A'..'Z' в C-локализации (§20.2.1, §21.7)
int isupper(int);   // буквы в верхнем регистре: 'A'..'Z' в C-локализации (§20.2.1, §21.7)
int islower(int);   // буквы в нижнем регистре: 'a'..'z' в C-локализации (§20.2.1, §21.7)
int isdigit(int);   // десятичные цифры:'0'..'9'
int isxdigit(int);  // шестнадцатеричные цифры:'0'..'9' или 'a'..'f' или 'A'..'F'
int isspace(int);   // символы-разделители
int iscntrl(int);   // управляющие символы (ASCII 0.31 и 127)
int ispunct(int);   // пунктуация: ничего из вышеперечисленного
int isalnum(int);   // isalpha() | isdigit()
int isprint(int);   // доступное для печати
int isgraph(int),   // isalpha() | isdigit() | ispunct()

int toupper(int c); // эквивалент с в верхнем регистре
int tolower(int c); // эквивалент с в нижнем регистре
```

Реализации этих функций обычно используют символ в качестве индекса в таблице символьных атрибутов. Поэтому следующий код

```
if(('a'<=c && c<='z') || ('A'<=c && c<='Z'))   // алфавитный символ
{
  // ...
}
```

не только неудобоварим и подвержен ошибкам (на машинах с EBCDIC символьным набором он пропустит неалфавитные символы), но и неэффективен.

Перечисленные функции имеют аргумент типа ***int***, и передаваемое им целое должно быть представимо как ***unsigned char*** или ***EOF*** (чаще всего равно -***1***). Это вызывает проблемы в системах, где ***char*** является знаковым типом (см. §20.6[11]).

Эквивалентные функции для «широких» символов находятся в <*cwctype*> и <*wctype.h*>.

20.5. Советы

1. Предпочитайте строки типа ***string*** С-строкам; §20.4.1.
2. Используйте тип ***string*** для типа переменных или полей классов, но не в качестве базового класса; §20.3, §25.2.1.
3. Можно передавать аргументы типа ***string*** по значению и осуществлять возврат этого же типа по значению, предоставив системе улаживать проблемы с памятью; §20.3.6.
4. Когда нужна проверка диапазона символов применяйте ***at***(), а не индексирование или итераторы; §20.3.2, §20.3.5.
5. Для повышения быстродействия применяйте индексирование или итераторы, а не ***at***(); §20.3.2, §20.3.5.
6. Для чтения подстрок применяйте (прямо или косвенно) ***substr***(), а для записи в подстроки — ***replace***(); §20.3.12, §20.3.13.
7. Для локализации заданных значений внутри строк применяйте ***find***() (а не пишите явные циклы); §20.3.11.
8. Для эффективного добавления символов к строке применяйте ***append***(); §20.3.9.
9. Если быстродействие не критично, используйте для символьного ввода строки типа ***string***; §20.3.15.
10. Используйте ***string::npos*** для индикации «остаток строки»; §20.3.5.
11. При необходимости реализуйте классы интенсивно нагруженных строк с помощью низкоуровневых операций (а не разбрасывайте по программе низкоуровневые структуры данных); §20.3.10.
12. Используя строки типа ***string***, по необходимости перехватывайте исключения ***length_error*** и ***out_of_range***; §20.3.5.
13. Следите за тем, чтобы не передать указатель типа ***char**** со значением ***0*** строковой функции; §20.3.7.
14. При необходимости получения С-строки из ***string*** применяйте ***c_str***(); §19.4.6.
15. Вместо написания собственного кода используйте для проверки типа символов библиотечные функции ***isalpha***(), ***isdigit***() и т.д.; §20.4.2.

20.6. Упражнения

1. (*2) Напишите функцию, принимающую в качестве аргументов две строки типа ***string***, и возвращающую конкатенацию этих строк с точкой посредине. Например, для ***file*** и ***write*** функция должна вернуть ***file.write***. Напишите такую же функцию для работы с С-строками, опираясь только на библиотечные функции языка С, такие как ***malloc***`()` или ***strlen***`()`. Сравните две версии. Какие могут быть разумные критерии сравнения?
2. (*2) Перечислите различия между ***vector*** и ***basic_string***. Какие различия существенны?
3. (*2) Средства для работы со строками не во всем согласованы. Например, вы можете присвоить символ типа ***char*** строке типа ***string***, но вы не можете инициализировать ***string*** символом. Составьте список подобного рода рассогласований. Как их можно было бы устранить? Какие новые проблемы могут при этом возникнуть?
4. (*1.5) В классе ***basic_string*** множество функций-членов. Какие из них можно было бы определить глобально (вне класса), не потеряв в эффективности и удобстве записи?
5. (*1.5) Напишите версию ***back_inserter***`()` (§19.2.4), работающую с ***basic_string***.
6. (*2) Завершите ***Basic_substring*** из §20.3.13 и интегрируйте его с типом ***String***, который перегружает операцию `()` со смыслом «взять подстроку», а во всем остальном эквивалентен типу ***string***.
7. (*2.5) Напишите функцию ***find***`()`, находящую в строке первое вхождение регулярного выражения. Используйте `?` для обозначения «любого символа», `*` — для обозначения любого числа символов, не отвечающих следующей части регулярного выражения, и `[`***abc***`]` — для обозначения любого символа из тех, что указаны в квадратных скобках (здесь это ***a***, ***b*** и ***c***). Другие символы должны совпадать буквально. Например, ***find***`(`***s***`, "`***name***`:")` возвращает указатель на первое вхождение ***name***: в строку ***s***; ***find***`(`***s***`, "[`***nN***`]`***ame***`:")` возвращает указатель на первое вхождение в строку ***s*** либо ***name***:, либо ***Name***:; ***find***`(`***s***`, "[`***nN***`]`***ame***`(*)")` возвращает указатель на первое вхождение в строку ***s*** либо ***name***, либо ***Name*** с дальнейшими (возможно пустыми) последовательностями символов в круглых скобках.
8. (*2.5) Каких операций вам не хватает для работы с регулярными выражениями в примере §20.6[7]? Выявите их и добавьте. Какова наглядность ваших операций для работы с регулярными выражениями по сравнению с традиционными? Сравните скорость работы вашего решения по сравнению со стандартными средствами.
9. (*2.5) Воспользуйтесь библиотекой регулярных выражений для реализации операции поиска по образцу над классом ***String***, имеющим ассоциированный с ним класс ***Substring***.
10. (*2.5) Подумайте, как можно спроектировать идеальный класс для универсальной работы с текстами. Назовите его ***Text***. Какими он будет обладать возможностями? Какую нагрузку на реализацию вызовут ваши «идеальные» средства?

11. (*1.5) Определите перегруженные версии для ***isalpha***(), ***isdigit***() и т.д., чтобы они могли корректно работать с ***char***, ***unsigned char*** и ***signed char***.
12. (*2.5) Напишите класс ***String***, который оптимизирован для коротких строк, содержащих не более 8 символов. Сравните его по быстродействию со стандартным ***string*** и ***String*** из §11.12. Возможно ли написать комбинированную версию строкового класса, сочетающего преимущества класса коротких строк и общего класса строк?
13. (*2) Оцените скорость копирования строк типа ***string***. Как ваша конкретная реализация строкового класса оптимизирует копирование?
14. (*2.5) Сравните быстродействие трех функций ***complete_name***() из §20.3.9 и §20.3.10. Попробуйте сами написать функцию ***complete_name***(), работающую наиболее производительно. Ведите список ошибок, выявленных на этапах программирования и тестирования.
15. (*2.5) Представьте, что считывание строк средней длины (от5 до 25 символов) из потока ***cin*** — самое узкое место в вашей системе. Напишите функцию ввода, читающую такие строки с максимально возможной скоростью. Допустимо организовать интерфейс функции в угоду быстродействию (даже за счет некоторого неудобства в использовании). Сравните результат с реализацией операции >> для типа ***string***.
16. (*1.5) Напишите функцию ***itos***(***int***), возвращающую строковое представление целого числа.

21

Потоки

Что вы видите — то вы и получите.
— Брайан Керниган

Ввод и вывод — потоки ***ostream*** — вывод встроенных типов — вывод типов, определяемых пользователем — виртуальные функции вывода — потоки ***istream*** — ввод встроенных типов — неформатированный ввод — состояние потока — ввод типов, определяемых пользователем — исключения ввода/вывода — связывание потоков — «часовые» — форматированный вывод целых и чисел с плавающей запятой — поля и выравнивание — манипуляторы — стандартные манипуляторы — манипуляторы, определяемые пользователем — файловые потоки — закрытие потоков — строковые потоки — строковые буферы — буферы потоков — локализация — функции обратного вызова для потоков — семейство функций ***printf***() — советы — упражнения.

21.1. Введение

Проектирование и реализация средств универсального ввода/вывода для любого языка программирования является сложной задачей. Традиционно, средства ввода/вывода создавались исключительно для работы с небольшим числом встроенных типов данных. Но любая нетривиальная программа на C++ содержит множество пользовательских типов, и для них тоже нужно выполнять ввод/вывод. Средства ввода/вывода должны быть простыми, удобными и безопасными, эффективными и гибкими, и прежде всего — полными. Никто еще не нашел единственного решения, удовлетворяющего всех. Поэтому важно, чтобы у пользователя сохранялась возможность вырабатывать альтернативные средства ввода/вывода, а также приспосабливать стандартные средства для специфических нужд конкретных программ.

Язык C++ позволяет пользователю определять новые типы, столь же эффективные и удобные, как встроенные типы. Поэтому разумным требованием является реализация ввода/вывода языка C++ исключительно на базе средств, доступных каждому пользователю. Рассматриваемые в настоящем разделе средства потокового

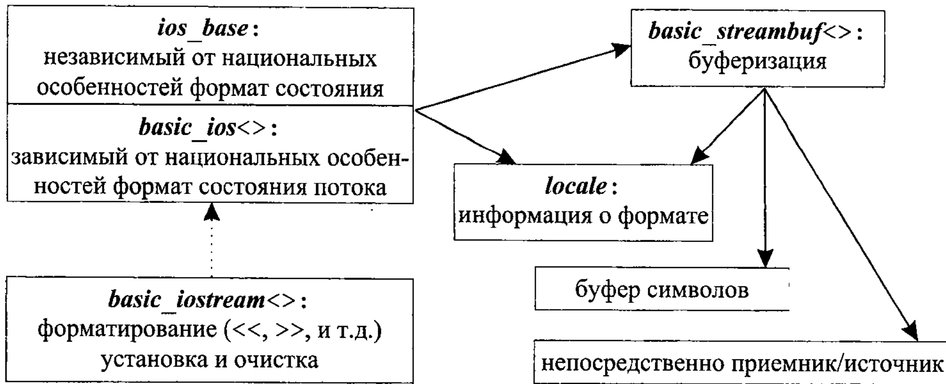
ввода/вывода языка C++ в значительной степени удовлетворяют сформулированному требованию:

- §21.2 *Вывод*: То, что разработчик программ считает выводом, на самом деле есть преобразование объектов некоторого типа, например ***int***, ***char**** или ***Employee_record***, в последовательность символов. Здесь рассматриваются средства записи встроенных и пользовательских типов в выходной поток.
- §21.3 *Ввод*: Здесь рассмотрены средства ввода символов, строк и других встроенных и пользовательских типов данных.
- §21.4 *Форматирование*: Часто на внешний вид вывода накладываются специальные требования. Например, значения типа ***int*** нужно выводить в десятичном виде, а указатели — в шестнадцатеричном; числа же с плавающей запятой часто требуется выводить с заданной степенью точности. Обсуждается управление форматированием и соответствующие приемы программирования.
- §21.5 *Файлы и потоки*: По умолчанию, каждая C++-программа может использовать стандартные потоки, такие как ***cout*** (вывод), ***cin*** (ввод) и ***cerr*** (поток ошибок). Чтобы использовались конкретные устройства и файлы, потоки должны к этим устройствам и файлам прикрепляться. Описываются механизмы открытия и закрытия файлов, а также прикрепление потоков к файлам и строкам ***string***.
- §21.6 *Буферизация*: Чтобы ввод/вывод был эффективным, нужно реализовать стратегию буферизации, удобную как для данных (читаемых или записываемых), так и для их источника/приемника. Описываются основные способы буферизации потоков.
- §21.7 *Локализация*: Объект ***locale*** специфицирует внешний вид выводимых чисел, какие символы считать буквами и т.д. Он инкапсулирует нюансы ввода/вывода, зависящие от национальных (локальных) особенностей. Последние используются системой ввода/вывода неявно и рассматриваются здесь очень кратко.
- §21.8 *Ввод/вывод в стиле языка C*: Обсуждается библиотечная функция языка C ***printf***() из заголовочного файла <***stdio.h***> и связь этой библиотеки с библиотекой C++ из <***iostream***>.

Знание средств реализации потоковой библиотеки не обязательно для ее успешного использования. Кроме того, реализация различается от системы к системе. В то же время, реализация ввода/вывода — задача не только сложная, но и интересная. Реализация ввода/вывода содержит интересные решения, пригодные для использования в иных сферах. Именно поэтому с технологиями, лежащими в основе реализации ввода/вывода, ознакомиться полезно.

В данной главе потоковая система ввода/вывода изучается ровно в той степени, которая позволит вам оценить ее структуру, применять ее на практике для самых общеупотребительных случаев, а также позволит вам расширять ее для работы с новыми пользовательскими типами данных. Если же вы хотите сами написать стандартный поток, ввести новый вид потока или в деталях учесть нюансы, зависящие от национальных языков, вам придется ознакомиться со стандартом, хорошим системным справочником и примерами работающего кода в добавок к тем, что представлены в данной главе.

Ключевые компоненты системы потокового ввода/вывода показаны на рисунке:



Пунктирная стрелка, идущая от ***basic_iostream***<>, означает, что ***basic_ios***<> является виртуальным базовым классом; сплошные стрелки соответствуют указателям. Классы, содержащие в имени угловые скобки, являются шаблонами, параметризуемыми символьным типом и содержащие национальные особенности представления символов.

Концепция потоков и присущая ей стандартная система обозначений приложимы к широкому кругу задач. Потоки применяются при передаче объектов между компьютерами (§25.4.1), при шифровании потока сообщений (§21.10[22]), для сжатия данных, для долговременного хранения объектов и для решения многих других задач. Но в данной главе обсуждение ограничивается *стандартным вводом/выводом, ориентированным на символы.*

Объявления классов потокового ввода/вывода и шаблонов (достаточные для обращения, но не для использования) и определения операторами ***typedef*** стандартных типов представлены в заголовочном файле <***iosfwd***>. Этот заголовочный файл требуется в тех случаях, когда вы хотите включить лишь часть из всех средств ввода/вывода.

21.2. Вывод

Однородный и безопасный подход как ко встроенным типам, так и к типам, определяемым пользователем, достигается за счет перегрузки функций вывода. Например:

```
put(cerr,"x=");     // cerr это поток вывода ошибок
put(cerr,x);
put(cerr,'\n');
```

Тип аргумента определяет выбор конкретной функции. Это решение хорошо известно (в том числе и в других языках программирования), но оно слегка тяжеловесное в связи с назойливым повтором имени функции. Перегрузка операции << (поместить в поток) существенно улучшает запись операции вывода и позволяет программисту осуществлять вывод последовательности объектов в рамках единственного оператора языка C++.

```
cerr << "x= "<<x<< '\n';
```

Если x есть целая переменная со значением *123*, этот оператор выведет

```
x = 123
```

за чем следует переход на новую строку в потоке *cerr*. Аналогично, если *x* имеет тип *complex* (§22.5) со значением (1,2.4), то будет следующий результат вывода в поток *cerr*:

```
x= (1,2.4)
```

Таким стилем записи можно пользоваться тогда, когда для типа определена операция <<, а пользователь всегда это может сделать для новых типов.

Операция вывода << нужна для того, чтобы избежать многословности, присущей именованным функциям вывода. Но почему именно <<? Новый лексический знак ввести в язык C++ нельзя (§11.2). Можно было бы использовать знак операции присваивания для обозначения ввода/вывода, но многие люди предпочитают иметь для этих операций разные знаки. Кроме того, операция = правоассоциативна, то есть *cout=a=b* означает *cout=(a=b)*, а не *(cout=a)=b* (§6.2). Я также пытался применить операции < и >, но их традиционный смысл «меньше чем» и «больше чем» так сильно застрял в головах людей, что применение этих значков в коде ввода/вывода было просто нечитаемым.

Операции << и >> не часто применяются ко встроенным типам, так что указанная проблема для этих операций не столь остра. Когда эти знаки операций применяются для ввода/вывода, я про себя читаю << как «записать в», а операцию >> — как «прочесть из». Люди с сильной склонностью к технической терминологии часто называют эти операции как *inserter* и *extractor*, соответственно[1]. Приоритет операций << и >> невысок, так что отсутствует нужда в дополнительных группирующих круглых скобках:

```
cout << "a*b+c=" << a*b+c << '\n';
```

Круглые скобки требуются лишь для выражений, содержащих операции с приоритетом, более низким, чем у операций << и >>. Например:

```
cout << "a^b|c=" << (a^b|c) << '\n';
```

Операция сдвига влево (§6.2.4) также может использоваться в выражениях вместе с операцией вывода, но конечно же, должна окружаться круглыми скобками:

```
cout<< "a<<b=" << (a<<b) << '\n';
```

21.2.1. Потоки вывода

Класс *ostream* реализует механизм преобразования значений различных типов в последовательность символов. Обычно эти символы затем выводятся с применением низкоуровневых операций. Есть много видов символов (§20.2), которые можно охарактеризовать с помощью *char_traits* (§20.2.1). Следовательно, *ostream* является специализацией общего шаблона *basic_ostream* для конкретного вида символов:

---

[1] На русском языке эти термины не устоялись, а называть их «инзертер» и «экстрактор» язык не поворачивается — да и необходимости в этом нет никакой. — *Прим. ред.*

```
template<class Ch, class Tr = char_traits<Ch> >
class std::basic_ostream : virtual public basic_ios<Ch, Tr>
{
public:
  virtual ~basic_ostream();
  // ...
};
```

Этот шаблон и соответствующие операции вывода определены в пространстве имен ***std*** и находятся в заголовочном файле <***ostream***>, содержащим те средства файла <***iostream***>, которые относятся к выводу.

Параметры шаблона ***basic_ostream*** контролируют тип символов, используемых в реализации; они не влияют на типы выводимых значений. Потоки, реализованные для обычного типа ***char***, и потоки «широких» символов, непосредственно поддерживаются каждой стандартной реализацией:

```
typedef basic_ostream<char> ostream;
typedef basic_ostream<wchar_t> wostream;
```

Для многих систем имеется возможность оптимизации вывода широких символов с помощью ***wostream*** до той степени, которую трудно достичь в потоках с единицей вывода в один байт.

Можно определить потоки, для которых физический ввод/вывод выполняется не в терминах символов. Однако такие потоки не соответствуют стандарту C++ и в данной книге не рассматриваются (§21.10[15]).

Базовый класс ***basic_ios*** представлен в заголовочном файле <***ios***>. Он управляет форматированием (§21.4), локализацией (§21.7) и доступом к буферам (§21.6). Он также определяет несколько типов ради удобства записи:

```
template<class Ch, class Tr = char_traits<Ch> >
class std::basic_ios : public ios_base
{
public:
  typedef Ch char_type;
  typedef Tr traits_type;
  typedef typename Tr::int_type int_type;    // тип целочисленного значения символов
  typedef typename Tr::pos_type pos_type;    // позиция в буфере
  typedef typename Tr::off_type off_type;    // смещение в буфере

  // см. также §21.3.3, §21.3.7, §21.4.4, §21.6.3, и §21.7.1 ...
};
```

Класс ***basic_ios*** запрещает копирование и создание объектов с помощью копирующего конструктора (§11.2.2). Соответственно, потоки ***ostream*** и ***istream*** нельзя копировать. Поэтому, если нужно изменить приемник потока, то вы можете либо изменить буфер (§21.6.4), либо перенаправить вывод при помощи указателя (§6.1.7).

Класс ***ios_base*** содержит информацию и операции, не зависящие от типа символов — например, точность для вывода чисел с плавающей запятой. Поэтому он не обязан быть шаблоном.

Кроме типов, определенных с помощью операторов ***typedef*** в ***ios_base***, библиотека потоков ввода/вывода использует знаковый интегральный тип ***streamsize*** для

представления размера буферов и числа символов, переданных в операциях ввода/вывода. Аналогично, с помощью оператора ***typedef*** определяется ***streamoff*** для представления смещений в потоках и буферах.

В файле <***iostream***> объявляется несколько стандартных потоков:

```
ostream cout;      // стандартный символьный поток вывода
ostream cerr;      // стандартный небуферированный поток вывода ошибок
ostream clog;      // стандартный поток вывода ошибок

wostream wcout;    // расширенный поток - аналог cout
wostream wcerr;    // расширенный поток - аналог cerr
wostream wclog;    // расширенный поток - аналог clog
```

Потоки ***cerr*** и ***clog*** имеют один и тот же адресат данных; они различаются лишь буферизацией. Поток ***cout*** пишет туда же, куда и ***stdout*** (§21.8), а потоки ***cerr*** и ***clog*** — туда же, куда и ***stderr***. Программист может создавать новые потоки по мере необходимости (§21.5).

21.2.2. Вывод встроенных типов

Класс ***ostream*** определяет операцию << для управления выводом встроенных типов:

```
template<class Ch, class Tr = char_traits<Ch> >
class basic_ostream : virtual public basic_ios<Ch, Tr>
{
public:
  // ...
  basic_ostream& operator<< (short n) ;
  basic_ostream& operator<< (int n) ;
  basic_ostream& operator<< (long n) ;

  basic_ostream& operator<< (unsigned short n) ;
  basic_ostream& operator<< (unsigned int n) ;
  basic_ostream& operator<< (unsigned long n) ;

  basic_ostream& operator<< (float f) ;
  basic_ostream& operator<< (double f) ;
  basic_ostream& operator<< (long double f) ;

  basic_ostream& operator<< (bool n) ;
  basic_ostream& operator<< (const void* p) ;     // пишет значение указателя

  basic_ostream& put (Ch c) ;                     // пишет c
  basic_ostream& write (const Ch*p, streamsize n) // p[0]..p[n-l]

  // ...
};
```

Функции ***put***() и ***write***() просто пишут символы в поток. Следовательно, операции << для вывода символов не обязательно делать функциями-членами. Их можно определить глобально:

```
template<class Ch, class Tr>
basic_ostream<Ch, Tr>& operator<< (basic_ostream<Ch, Tr>&, Ch) ;
template<class Ch, class Tr>
basic_ostream<Ch, Tr>& operator<< (basic_ostream<Ch, Tr>&, char) ;
```

```
template<class Tr>
basic_ostream<char,Tr>& operator<<(basic_ostream<char,Tr>&, char);
template<class Tr>
basic_ostream<char,Tr>& operator<<(basic_ostream<char,Tr>&, signed char);
template<class Tr>
basic_ostream<char,Tr>& operator<<(basic_ostream<char,Tr>&, unsigned char);
```

Аналогично определяется операция << для вывода символьных массивов с терминальным нулем:

```
template<class Ch, class Tr>
basic_ostream<Ch,Tr>& operator<<(basic_ostream<Ch,Tr>&, const Ch*);
template<class Ch, class Tr>
basic_ostream<Ch,Tr>& operator<<(basic_ostream<Ch,Tr>&, const char*);
template<class Tr>
basic_ostream<char,Tr>& operator<<(basic_osiream<char,Tr>&, const char*);
template<class Tr>
basic_ostream<char,Tr>& operator<<(basic_ostream<char,Tr>&, const signed char*);
template<class Tr>
basic_ostream<char,Tr>& operator<<(basic_ostream<char,Tr>&, const unsigned char*);
```

Операции вывода для типа ***string*** представлены в заголовочном файле ***<string>*** (см. также §20.3.15).

Функция ***operator***<<() возвращает ссылку на объект типа ***ostream***, для которого она была вызвана, так что к этому объекту можно снова применить операцию <<. Например:

```
cerr << "x = " << x;
```

где ***x*** имеет тип ***int***, интерпретируется следующим образом:

```
operator<<(cerr, "x= ").operator<<(x);
```

В частности, это означает, что когда несколько объектов выводятся в рамках одного оператора языка C++, они будут расположены в ожидаемом порядке — слева направо. Например:

```
void val(char c)
{
  cout<< "int('" << c << "') = " << int(c) << '\n';
}

int main()
{
  val('A');
  val('Z');
}
```

Для реализации с символами ASCII эта программа выведет:

```
int('A') = 65
int('Z') = 90
```

Отметим, что символьные литералы имеют тип ***char*** (§4.3.1), так что ***cout***<<'***Z***' выведет букву ***Z***, а не число ***90***.

По умолчанию тип ***bool*** выводится как ***0*** или ***1***. Если вам это не нравится, можете установить форматирующий флаг ***boolalpha*** из <***iomanip***> (§21.4.6.2) и получить ***false*** или ***true***. Например:

```
int main ()
{
  cout << true << ' ' << false << '\n' ;
  cout << boolalpha;      // использовать символьное представление для true и false
  cout << true << ' ' << false << '\n' ;
}
```

В результате вывод будет следующим:

```
1 0
true false
```

Более точно, флаг ***boolalpha*** обеспечивает текстовый вывод, зависящий от текущей локализации (национальных стандартов). Если я установлю нужную (датскую) локализацию (§21.7), то получу следующий вывод:

```
1 0
sandt falsk
```

Форматирование чисел с плавающей запятой, вывод целых чисел в разных системах счисления и т.д. рассматриваются в §21.4.

Функция ***ostream: : operator<< (const void*)*** выводит значение указателя в формате, зависяшем от машинной архитектуры. Например:

```
int main ()
{
  int* p = new int;
  cout << "local " << &p << ", free store " << p << '\n' ;
}
```

выведет на моей машине

```
local 0x7fffead0, free store 0x500c
```

В других системах вывод указателя может быть иным.

21.2.3. Вывод пользовательских типов

Рассмотрим пользовательский тип complex (§11.3):

```
class complex
{
public:
  double real () const {return re; }
  double imag () const {return im; }
  // ...
};
```

Операцию вывода << для типа ***complex*** определяем следующим образом:

```
ostream& operator<< (ostream& s, const complex& z)
{
  return s << '(' << z.real() << ',' <<z.imag() << ')';
}
```

Теперь для типа ***complex*** операцию << можно использовать точно так же, как для встроенных типов:

```
int main()
{
  complex x(1,2);
  cout << "x= " << x << '\n';
}
```

На выходе получаем

```
x= (1,2)
```

Отметим, что при этом не требуется изменений в объявлении класса ***ostream***. Это очень хорошо, поскольку не будут вноситься изменения в заголовочный файл <***ostream***>. Запрет на внесение изменений в этот файл гарантирует от порчи его структур данных и позволяет осуществлять модификацию этого файла без влияния на код пользователя.

21.2.3.1. Виртуальные функции вывода

Функции-члены класса ostream не виртуальные. Операции вывода, добавляемые пользователем, не являются функциями-членами, и, значит, они тоже не виртуальные. Одна из причин такого положения дел состоит в желании достичь максимальной эффективности для операции помещения символа в буфер. Это то место, где критически важна производительность, и поэтому нужно обеспечить встраивание функций. А виртуальные функции применяются для гибкости операций, имеющих дело с переполнением (и переопустошением) буфера (§21.6.4).

Тем не менее, иногда программисту нужно вывести объект, для которого известен лишь базовый класс. Поскольку точный тип неизвестен, корректность вывода недостижима путем определения операции << для всех новых типов. Вместо этого в абстрактном базовом классе можно определить виртуальную функцию вывода:

```
class My_base
{
public:
  //...
  virtual ostream& put(ostream& s) const = 0;    // пишет *this в s
};

ostream& operator<< (ostream& s, const My_base& r)
{
  return r.put(s);                                // используется правильная put()
}
```

Здесь функция ***put***() является виртуальной функцией, обеспечивающей корректность вывода при использовании операции <<.

Теперь мы можем написать:

```
class Sometype : public My_base
{
public:
  // ...
  ostream& put(ostream& s) const;        // замещает My_base::put()
};

void f(const My_base& r, Sometype& s) // используется <<, вызывающая правильную put()
{
  cout << r << s;
}
```

Таким образом виртуальная функция ***put***() интегрируется в программный каркас, формируемый классом ostream и операциями <<. Это полезный универсальный прием для введения операций, которые ведут себя как виртуальные функции с выбором на этапе выполнения на основе их второго аргумента.

21.3. Ввод

Ввод обрабатывается почти так же, как вывод. Имеется класс ***istream***, который предоставляет операцию >> («прочесть из») для небольшого набора стандартных типов. Можно определять эту операцию для любых пользовательских типов.

21.3.1. Потоки ввода

Аналогично классу ***basic_ostream*** (§21.2.1) класс ***basic_istream*** определен в заголовочном файле <***istream***>, содержащем те средства файла <***iostream***>, которые относятся к вводу:

```
template<class Ch, class Tr = char_traits<Ch> >
class std::basic_istream : virtual public basic_ios<Ch, Tr>
{
public:
  virtual ~basic_stream();
  // ...
};
```

Базовый класс ***basic_ios*** описан в §21.2.1.

Два стандартных потока ввода ***cin*** и ***wcin*** приведены в <***iostream***>:

```
typedef basic_istream<char> istream;
typedef basic_istream<wchar_t> wistream;

istream cin;     // стандартный поток ввода символов типа char
wistream wcin;   // стандартный поток ввода символов типа wchar_t
```

Поток ***cin*** читает из того же источника, что и ***stdin*** (§21.8).

21.3.2. Ввод встроенных типов

Класс *istream* определяет операцию >> для встроенных типов:

```
template<class Ch, class Tr = char_traits<Ch> >
class basic_istream: virtual public basic_ios<Ch,Tr>
{
public:
  // ...
  // форматированный ввод:
  basic_istream& operator>> (short& n) ;
  basic_istream& operator>> (int& n) ;
  basic_istream& operator>> (long& n) ;

  basic_istream& operator>> (unsigned short& u) ;
  basic_istream& operator>> (unsigned int& u) ;
  basic_istream& operator>> (unsigned long& u) ;

  basic_i$tream& operator>> (float&f) ;
  basic_istream& operator>> (double& f) ;
  basic_istream&.operator>> (long double& f) ;

  basic_istream& operator>> (bool& b) ;
  basic_istream& operator>> (void* & p) ;
  // ...
};
```

Функции *operator*>>() определяются в следующем стиле:

```
istream& istream::operator>> (T& tvar)
// T - это тип, для которого объявляется istream::operator>>
{
  // пропустить пробельные символы, затем прочитать значение типа T в tvar
  return *this;
}
```

Поскольку операция >> пропускает пробельные символы, вы можете читать последовательность целых чисел, разделенных такими символами, следующим образом:

```
int read_ints (vector<int>& v)     // заполнение v, возврат числа прочитанных int
{
  int i = 0;
  while (i<v.size() && cin>>v[i]) i++;
  return i;
}
```

Нецелые на входе приведут к окончанию чтения. Например, если на вход поступят

1 2 3 4 5.6 7 8.

функция *read_ints*() прочтет пять целых чисел:

1 2 3 4 5

и оставит точку для последующего чтения. Пробельные символы соответствуют таковым в языке С (пробел, табуляция, новая строка, новая страница и возврат каретки) и распознаются функцией *isspace*(), определенной в <*cctype*> (§20.4.2).

По умолчанию, операция >> пропускает пробельные символы. Но мы можем изменить такое поведение: *is.unsetf(ios_base::skipws)* заставит операцию >> для *is* трактовать пробельные символы так же, как обычные символы (см. §21.4.1, §21.4.6 и §21.4.6.2).

Применяя *istream*, часто совершают следующую типичную ошибку: не замечают, что фактически введено было не то, что ожидалось (например, из-за подачи на вход потока символ неожиданного формата). Нужно или проверять состояние потока ввода (§21.3.3) перед тем, как использовать то, что было предположительно введено, или задействовать исключения (§21.3.6).

Ожидаемый на входе формат определяется текущими установками локализации (§21.7). По умолчанию, логические значения true и false представляются как *1* и *0*, соответственно. Целые числа должны представляться в десятичном формате, а числа с плавающей запятой должны иметь тот формат, какой принят в языке C++. Установив *basefield* (§21.4.2), можно прочесть *0123* как восьмеричное число с десятичным значением *83*, а *0xff* как шестнадцатеричное число с десятичным значением *255*. Формат, по которому читаются значения для указателей, целиком зависит от конкретной реализации (проверьте, как это делается в вашей реализации).

Удивительно, но для чтения символов функции-члена, соответствующей операции >>, нет. Зато для этого есть функция-член *get()* (§21.3.4), и поэтому нет никакой необходимости определять такую операцию >> в качестве функции-члена. Из потока мы можем прочитать символ в связанный с потоком символьный тип. Если это *char*, то мы можем прочесть символ и в *signed char*, и в *unsigned char*:

```
template<class Ch, class Tr>
basic_istream<Ch,Tr>& operator>> (basic_istream<Ch,Tr>&, Ch&);

template<class Tr>
basic_istream<char,Tr>& operator>> (basic_istream<char,Tr>&, unsigned char&);

template<class Tr>
basic_istream<char,Tr>& operator>> (basic_istream<char,Tr>&, signed char&);
```

С точки зрения пользователя не важно, определена ли операция >> с помощью функции-члена, или нет.

Как и в остальных операциях >>, здесь также сначала пропускаются пробельные символы. Например:

```
void f()
{
  char c;
  cin >> c;
  // ...
}
```

Таким образом, в переменную с попадет первый символ из *cin*, не являющийся разделительным (пробельным) символом.

Кроме того, можно читать и в массив символов:

```
template<class Ch, class Tr>
basic_istream<Ch,Tr>& operator>> (basic_istream<Ch,Tr>&, Ch*);

template<class Tr>
basic_istream<char,Tr>& operator>> (basic_istream<char,Tr>&, unsigned char*);

template<class Tr>
basic_istream<char,Tr>& operator>> (basic_istream<char,Tr>&, signed char*);
```

Эти операции также пропускают пробельные символы. Затем они считывают символы из потока в массив-операнд до тех пор, пока не встретится пробельный символ или конец файла. В конце концов, они завершают строку нулем. Ясно, что это открывает лазейку для переполнения, так что лучше читать в ***string*** (§20.3.15). В то же время, можно ограничить число читаемых операцией >> символов: ***is.width(n)*** устанавливает, что последующая операция >> над ***is*** прочтет в массив максимум ***n-1*** символ. Например:

```
void g()
{
  char v[4];
  cin.width(4);
  cin >> v;
  cout<< "v = " << v << endl;
}
```

Здесь в ***v*** будет прочитано самое большее три символа и завершающий нуль.

Вызов функции ***width()*** для потока ***istream*** влияет лишь на следующую за этим операцию чтения >> в массив, и не влияет на считывание в иные типы переменных.

21.3.3. Состояние потока

Каждый поток (***istream*** или ***ostream***) характеризуется своим *состоянием* (*state*). Проверкой состояния потока можно выловить ошибки и нестандартные положения.

С состоянием потока работает класс ***basic_ios*** из <***ios***>, базовый для класса ***basic_istream***:

```
template<class Ch, class Tr = char_traits<Ch> >
class basic_ios: public ios_base
{
public:
  //...
  bool good() const;                    // след. операция может быть выполнена
  bool eof() const;                     // виден конец ввода
  bool fail() const;                    // след. операция не может быть выполнена
  bool bad() const;                     // поток испорчен

  iostate rdstate() const;              // получить флаги состояния ввода/вывода
  void clear(iostate f = goodbit);      // установить флаги состояния i/o
  void setstate(iostate f) {clear(rdstate() |f); }   // добавить f к флагам состояния i/o

  operator void*() const;               // не нуль, если !fail()
  bool operator!() const {return fail(); }
  //...
};
```

Если ***good()*** возвращает ***true***, то предыдущая операция ввода завершилась успешно и может выполняться следующая операция ввода; в противном случае следующая операция ввода не пройдет — это просто нулевая операция, так что попытка ввода в переменную ***v*** оставляет переменную ***v*** без изменения (если ***v*** имеет тип, для которого имеются функции-члены классов ***istream*** и ***ostream***). Разница между состояниями ***fail()*** и ***bad()*** невелика — в первом случае считается, что поток не ис-

порчен (возможно просто потеряны символы), во втором же случае ничего определенного сказать нельзя.

Состояние потока представляется набором флагов. Как и большинство констант, характеризующих поведение потоков, эти флаги определяются в ***ios_base*** — базовом классе для ***basic_ios***:

```
class ios_base
{
public:
  // ...
  typedef implementation_defined2 iostate;

  static const iostate badbit,    // поток испорчен
                       eofbit,    // виден конец файла
                       failbit,   // следующая операция не будет выполнена
                       goodbit;   // goodbit==0
  // ...
};
```

Флагами состояния ввода/вывода можно манипулировать непосредственно. Например:

```
void f()
{
  ios_base::iostate s = cin.rdstate(); // возвращает набор битов состояния ввода/вывода

  if(s & ios_base::badbit)
  {
    // возможно, при вводе потеряны символы
  }
  // ...
  cin.setstate(ios_base::failbit);
  // ...
}
```

Когда поток используется в качестве условия, состояние потока проверяется функциями ***operator void****() или ***operator***!(). Эти проверки возвращают ***true*** только тогда, когда состояние потока есть !***fail***() или ***fail***(), соответственно. Например, универсальную функцию копирования можно написать так:

```
template<class T> void iocopy(istream& is, ostream& os)
{
  T buf;
  while(is>>buf) os << buf<< '\n';
}
```

Выражение ***is>>buf*** возвращает ссылку на ***is***, и состояние последней проверяется неявным вызовом ***istream::operator void****(). Функцию ***iocopy***() можно использовать следующим образом:

```
void f(istream& i1, istream& i2, istream& i3, istream& i4)
{
  iocopy<complex>(i1,cout);   // копирование чисел типа complex
  iocopy<double>(i2,cout);    // копирование чисел типа double
```

```
    iocopy<char> (i3, cout) ;      // копирование чисел типа char
    iocopy<string> (i4, cout) ;    // копирование слов, разделенных пробельными символами
}
```

21.3.4. Ввод символов

Операция >> предназначена для форматированного ввода, то есть для чтения объектов ожидаемого типа и формата. Когда мы хотим читать символы без предположений об их смысле, мы используем *функции неформатированного ввода* (*unformatted input functions*):

```
template<class Ch, class Tr = char_traits<Ch> >
class basic_istream : virtual public basic_ios<Ch, Tr>
{
public:
    // ...
    // неформатированный ввод:
    streamsize gcount () const;   // число символов, прочитанных последней функцией
                                  // неформатированного ввода

    int_type get () ;                          // чтение одного Ch (или Tr::eof())

    basic_istream& get (Ch& c) ;               // чтение одного Ch в с

    basic_istream& get (Ch* p, streamsize n) ;        // терминатор - новая строка
    basic_islream& get (Ch* p, streamsize n, Ch term) ;

    basic_istream& getline (Ch* p, streamsize n) ; // терминатор - новая строка
    basic_istream& getline (Ch* p, streamsize n, Ch term) ;

    basic_istream& ignore (streamsize n = l, int_type t = Tr: :eof () ) ;
    basic_istream& read (Ch* p, streamsize n) ;    // чтение не более n символов
    // ...
};
```

Кроме того, в заголовочном файле <*string*> имеется функция ***getline*** () для стандартного типа ***string*** (§20.3.15).

Функции неформатированного ввода не пропускают пробельных символов.

Если функции ***get*** () или ***getline*** () не читают и не удаляют из потока ни одного символа, вызывается ***setstate*** (***failbit***) и все последующие попытки чтения из потока закончатся неудачей (или будет сгенерировано исключение; §21.3.6).

Функция ***get*** (***char***&) считывает один символ в свой аргумент. Например:

```
int main ()
{
    char c;
    while (cin.get (c) ) cout.put (c) ;
}
```

Функция с тремя аргументами ***s.get*** (***p, n, term***) читает не более ***n-1*** символов в ***p*** [***0***] . . ***p*** [***n-2***]. Вызов ***get*** () всегда помещает ***0*** после помещенных в массив ***p*** [] символов, так что ***p*** должен указывать на массив, содержащий не менее ***n*** символов. Третий аргумент, ***term***, задает терминальный символ. Типичное применение трехаргументной функции ***get*** () — считывание физической строки текста (ограничен-

ной, например, терминальным символом '\n') в буфер фиксированного размера для ее последующего анализа:

```
void f()
{
  char buf[100];
  cin >> buf;                  // подозрение: когда-нибудь приведет к переполнению
  cin.get(buf,100, '\n');      // безопасно
  //...
}
```

Если терминальный символ обнаружен, он остается первым непрочитанным символом в потоке. Никогда не вызывайте функцию ***get***() дважды подряд без удаления терминального символа, как в следующем примере:

```
void subtle_error()
{
  char buf[256];

  while (cin)
  {
    cin.get(buf,256);     // читаем строку
    cout << buf;          // печатаем строку
    // Oops: забыли удалить '\n' из cin - следующий get() не сработает
  }
}
```

Отсюда следует, что лучше использовать ***getline***() вместо ***get***(). Функция ***getline***() во всем аналогична ***get***(), но она удаляет терминальный символ из потока ***istream***. Например:

```
void f()
{
  char word[MAX_WORD] [MAX_LINE];     // MAX_WORD массив из MAX_LINE
                                      // символов в каждом

  int i = 0;
  while(cin.getline(word[i++],MAX_LINE,'\n') && i<MAX_WORD);
  //...
}
```

Когда эффективность не является важнейшей целью, лучше считывать символы в строку типа ***string*** (§3.6, §20.3.15), ибо в этом случае отпадают проблемы с распределением памяти и переполнением буфера. Функции ***get***(), ***getline***() и ***read***() нужны для высочайшей производительности, за которую приходится платить запутанным интерфейсом. При этом также отпадает необходимость в повторном чтении для того, чтобы выяснить причины окончания ввода, а также имеется возможность надежно ограничить ввод символов заданным их числом.

Вызов ***read***(***p***,***n***) читает не более ***n*** символов в ***p***[***0***]..***p***[***n-1***]. Эта функция не полагается на терминальный символ и не добавляет нуль после помещенных в массив символов. Как следствие, она действительно считывает ***n*** символов (а не ***n-1***). Другими словами, она просто читает символы и не пытается превратить считанное в С-строку.

Функция *ignore*() читает символы так же, как функция *read*(), но нигде не хранит их. Как и *read*(), она читает ровно *n* символов. Вызов функции *ignore*() без аргументов приводит к чтению (отбрасыванию) одного (следующего) символа. Как и функция *getline*(), функция *read*() имеет необязательный параметр, означающий терминальный символ, который она и отбрасывает, если добирается до него. Отметим, что по умолчанию терминальным символом для функции *ignore*() служит «конец файла».

Для всех перечисленных функций неформатированного ввода совсем не очевидно, что именно прекращает процесс ввода, и каков конкретный критерий завершения их работы. Однако всегда имеется возможность выяснить, не достигли ли мы конца файла (§21.3.3). Кроме того, имеется функция *gcount*(), возвращающая число символов, считанных из потока последним вызовом функции неформатированного ввода. Например:

```
void read_a_line(int max)
{
  // ...
  if(cin.fail())              // Oops: неправильный формат ввода
  {
    cin.clear();              // очистить флаги ввода (§21.3.3)
    cin.ignore(max, ';');     // пропуск до точки с запятой

    if(!cin)
    {
      // oops: достигли конца потока
    }
    else if(cin.gcount()==max)
    {
      // oops: прочли максимальное количество символов
    }
    else
    {
      // нашли и удалили точку с запятой
    }
  }
}
```

К сожалению, если прочитано максимальное количество символов, то уже невозможно установить, был ли достигнут терминальный символ (в качестве последнего символа).

Функция *get*() без аргумента эквивалентна функции *getchar*() из <*cstdio*> (§21.8). Она просто читает символ и возвращает его численное значение. Таким образом, она избегает делать предположения о том, какой использовался символьный тип. Если нет никакого символа, то *get*() возвращает подходящий маркер «конца файла» (то есть *traits_type*::*eof*() этого потока) и устанавливает поток в состояние *eof* (§21.3.3). Например:

```
void f(unsigned char* p)
{
  int i;
  while((i = cin.get() && i!=EOF)
```

```
    {
      *p++ = i;
      //...
    }
}
```

Здесь EOF — это возврат функции ***eof***() из обычных ***char_traits*** для типа ***char***. ***EOF*** представлен в заголовочном файле <***iostream***>. Таким образом, вместо цикла можно было бы сделать вызов ***read***(***p***, ***MAX_INT***), но мы написали явный цикл для того, чтобы, например, просматривать каждый символ по мере поступления такового. Уже говорилось, что сила языка C заключается в умении считать символ и принять решение (причем быстро), что с ним ничего не надо делать. Это важное достоинство, которое нельзя недооценивать, и в языке C++ оно сохранено.

В стандартном заголовочном файле <***cctype***> определено несколько функций, полезных при обработке ввода (§20.4.2). Например, функция ***eatwhite***(), считывающая пробельные символы, могла быть определена следующим образом:

```
istream& eatwhite (istream& is)
{
  char c;
  while (is.get(c))
  {
    if(!isspace(c))          // c - это пробельный символ?
    {
      is.putback(c);         // помещаем c назад в буфер ввода
      break;
    }
  }
  return is;
}
```

Вызов ***is.putback***(***c***) превращает ***c*** в следующий символ, читаемый из потока ***is*** (§21.6.4).

21.3.5. Ввод пользовательских типов

Как и в случае вывода, для пользовательских типов данных можно определять операции ввода. Однако для операций ввода важно, чтобы их второй аргумент не был ***константной*** ссылкой. Например:

```
istream& operator>> (istream& s, complex& a)
/*
   форматы ввода для complex ("f" означает число с плавающей запятой):
   f или (f) или (f,f)
*/
{
  double re = 0, im = 0;
  char c = 0;

  s >> c;
  if(c == '(')
  {
```

```
    s >> re >> c;
    if(c == ',') s >> im >> c;
    if(c != ')') s.clear(ios_base::failbit);
  }
  else
  {
    s.putback(c);
    s >> re;
  }

  if(s) a = complex(re,im);
  return s;
}
```

Несмотря на краткость кода обработки ошибок, он в состоянии выловить большинство их видов. Локальная переменная с инициализируется во избежание случайных совпадений ее значения с '(' после первой неудавшейся операции >>. Заключительная проверка состояния потока гарантирует, что переменная а изменяется лишь в том случае, если все прошло успешно.

Если обнаруживается ошибка форматирования, состояние потока устанавливается в ***failbit***. Состояние потока не устанавливается в ***badbit***, потому что поток сам по себе не разрушен. Пользователь может восстановить состояние потока (используя ***clear***()) и, возможно, обойти проблему и извлечь полезные данные из потока.

Операция по восстановлению состояния потока названа ***clear***(), поскольку ее типичное назначение — сброс состояние потока в ***good***(); ***ios_base::goodbit*** — умолчательное значение аргумента для функции ***clear***() (§21.3.3).

21.3.6. Исключения

Проверять наличие ошибок после каждой операции ввода/вывода неудобно, и поэтому основная причина возникновения таких ошибок заключается в том, что отсутствуют необходимые проверки в важных местах программы. В частности, типичным является отсутствие проверки наличия ошибок при выводе, хотя они встречаются и в этом случае.

Единственной функцией, непосредственно изменяющей состояние потока, является ***clear***(). Поэтому единственный способ заметить изменение состояния потока — это попросить функцию ***clear***() генерировать исключения. Функция-член ***exceptions***() класса ***basic_ios*** как раз и делает это:

```
template<class Ch, class Tr = char_traits<Ch> >
class basic_ios: public ios_base
{
public:
  // ...
  class failure;                          // класс исключений (см. §14.10)

  iostate exceptions() const;             // получить состояние исключений
  void exceptions(iostate except);        // установить состояние исключения
  // ...
};
```

Например:

```
cout.exceptions(ios_base::badbit|ios_base::failbit|ios_base::eofbit);
```

требует, чтобы ***clear***() генерировала исключение ***ios_base::failure***, если поток ***cout*** переходит в состояние ***bad***, ***fail*** или ***eof*** — другими словами, если любая операция вывода не выполнилась безупречно. Если необходимо, можно проверить ***cout*** с тем, чтобы выяснить в точности, что произошло. Аналогично,

```
cin.exceptions(ios_base::badbit|ios_base::failbit);
```

позволит перехватить не слишком редкий случай, когда на входе присутствуют данные не в ожидаемом формате и операция ввода не возвращает никаких значений из потока.

Вызов ***exceptions***() без аргументов возвращает набор флагов состояния ввода/вывода, инициировавшего исключение. Например:

```
void print_exceptions(ios_base& ios)
{
  ios_base::iostate s = ios.exceptions();
  if(s&ios_base::badbit) cout << "throws for bad";
  if(s&ios_base::failbit) cout << "throws for fail";
  if(s&ios_base::eofbit) cout << "throws for eof";
  if(s == 0) cout << "doesn't throw";
}
```

Главная цель исключений ввода/вывода — отлавливать наименее вероятные ошибки, о которых часто забывают. Другая цель — управлять вводом/выводом. Например:

```
void readints(vector<int>& s)
{
  ios_base::iostate old_state = cin.exceptions();   // сохраняем состояние исключения
  cin.exceptions(ios_base::eofbit);                 // генерируем исключение для eof

  for(;;)
    try
    {
      int i;
      cin>>i;
      s.push_back(i);
    }
    catch(ios_base::failure)    // ok: достигнут конец файла
    {
      break;
    }

  cin.exceptions(old_state);    // восстанавливаем состояние исключения
}
```

Вопросы, задаваемые в связи с этим таковы: «Разве это ошибка?» или «Разве это действительно исключительная ситуация?» (§14.5). Я считаю, что ответ на оба этих вопроса — «нет». И поэтому я предпочитаю работать с состоянием потока напрямую. То, что легко обрабатывается локальными управляющими структурами внутри функций, редко когда нуждается в использовании исключений.

21.3.7. Связывание потоков

Функция *tie*() из *basic_ios* используется для того, чтобы устанавливать и разрывать связи между *istream* и *ostream*:

```
template<class Ch, class Tr = char_traits<Ch> >
class std::basic_ios: public ios_base
{
  // ...
  basic_ostream<Ch,Tr>* tie() const;     // получаем указатель на связанный поток
  basic_ostream<Ch,Tr>* tie(basic_ostream<Ch,Tr>* s); // привязываем *this к s
  // ...
};
```

Рассмотрим следующий код:

```
string get_passwd()
{
  string s;
  cout << "Password: ";
  cin >> s;
  // ...
}
```

Как мы можем быть уверены в том, что надпись *Password*: появится на экране до того, как завершится операция чтения? Вывод в *cout* буферизуется, так что если бы *cin* и *cout* были независимыми, то надпись *Password*: не появилась бы на экране до тех пор, пока буфер полностью не заполнился бы. Решение этой проблемы состоит в том, что поток *cout* связывается с потоком *cin* операцией *cin.tie*(&*cout*).

Когда *ostream* связан с *istream*, поток ostream очищается каждый раз, когда операция ввода над *istream* испытывает недостаток символов, то есть когда требуются символы из конечного источника ввода для завершения операции ввода. Таким образом,

```
cout << "Password: ";
cin >> s;
```

эквивалентно

```
cout << "Password: ";
cout.flush();
cin >> s;
```

Поток ввода может иметь не более одного связанного с ним *ostream*. Вызов *s.tie*(*0*) отвязывает поток *s* от потока, с которым он был связан (если был связан). Как и большинство других поточных функций, устанавливающих значение, функция *tie*(*s*) возвращает прежнее значение, то есть она возвращает предыдущий связанный поток или *0*. При вызове без аргумента функция *tie*() возвращает текущее значение без его изменения.

Из стандартных потоков *cout* привязывается к *cin*, а *wcout* — к *wcin*. Потоки *cerr* не нуждаются в привязке, поскольку они не буферизованы, а потоки *clog* не предназначены для взаимодействия с пользователем.

21.3.8. Часовые (sentries)

Когда я писал операции << и >> для типа ***complex***, я не беспокоился о связывании потоков (§21.3.7) и не думал о том, генерируются ли исключения при изменении состояний потоков (§21.3.6). Я предполагал (совершенно справедливо), что библиотечные функции сами об этом позаботятся. Но как? Всего таких функций несколько десятков, и если бы нам самим пришлось писать нетривиальный код для управления связанными потоками, *локализацией* (§21.7), исключениями и т.д., наш код стал бы слишком сложным и запутанным.

В стандартной библиотеке использован подход, предполагающий применение класса ***sentry*** («часовой») для реализации перечисленных выше общих задач. Код программы, который должен исполняться первым (префикс-код) — например, сброс связанного потока — предоставляется посредством конструктора класса ***sentry***. А код, который должен исполняться последним (суффикс-код) — например, генерация исключений из-за изменения состояния потока — предоставляется посредством деструктора класса ***sentry***:

```
template<class Ch, class Tr = char_traits<Ch> >
class basic_ostream : virtual public basic_ios<Ch, Tr>
{
  // ...
  class sentry;
  // ...
};

template<class Ch, class Tr = char_traits<Ch> >
class basic_ostream<Ch, Tr>: :sentry
{
public:
  explicit sentry (basic_ostream<Ch, Tr>& s) ;
  ~sentry () ;

  operator bool () ;
  // ...
};
```

Таким способом вычленяется общий код и остальные функции пишутся следующим образом:

```
template<class Ch, class Tr = char_traits<Ch> >
basic_ostream<Ch, Tr>& basic_ostream<Ch, Tr>: :operator<< (int i)
{
  sentry s (*this) ;

  if ( !s)                    // проверка готовности к началу вывода
  {
    setstate (failbit) ;
    return *this;
  }

  // выводим int
  return *this;
}
```

Рассмотренная технология предоставления префикс-кода и суффикс-кода в виде конструктора и деструктора специального класса полезна и во многих других контекстах.

Само собой разумеется, что и класс ***basic_istream*** располагает своим собственным «часовым».

21.4. Форматирование

Все примеры из §21.2 относятся к так называемому неформатированному выводу, означающему, что объект превращается в последовательность символов по умолчательным правилам. Но часто возникает необходимость в более детализированном управлении этим процессом, например, в контроле за объемом памяти, выделяемом для операции вывода, или в задании формата вывода чисел. Аналогично, возникает необходимость и в управлении отдельными аспектами ввода.

Управление форматированием ввода/вывода сосредоточено в классе ***basic_ios*** и в его базовом классе ***ios_base***. Например, класс ***basic_ios*** содержит информацию о системе счисления (восьмеричная, десятичная или шестнадцатеричная), которая используется при вводе или выводе целых чисел. Он также содержит функции для установки и проверки этих управляющих переменных потока.

Класс ***basic_ios*** является базовым для ***basic_istream*** и ***basic_ostream***, так что управление форматом выполняется по отдельности для каждого потока.

21.4.1. Состояние форматирования

Форматирование ввода/вывода контролируется набором флагов и целочисленных значений в классе ***ios_base***:

```
class ios_base
{
public:
  // ...
  // имена флагов форматирования:
  typedef implementation_definedl fmtflags;

  static const fmtflags
          skipws,      // пропускать пробельные символы на входе
          left,        // выравнивание поля: заполнитель после значения
          right,       // заполнитель перед значением
          internal,    // заполнитель между знаком и значением
          boolalpha,   // использовать символьное представление для true и false
          dec,         // основание для вывода целых: 10
          hex,         // вывод по основанию 16 (шестнадцатеричный)
          oct,         // вывод по основанию 8 (восьмеричный)
          scientific,  // форма чисел с плавающей запятой: d.ddddddEdd
          fixed,       // dddd.dd
          showbase,    // префикс 0 для восьмеричных и 0x - для шестнадцатеричных
          showpoint,   // печатать хвостовые (незначащие) нули
          showpos,     // явный '+' для положительных int
          uppercase,   // 'E', 'X' а не 'e', 'x'
          adjustfield, // флаги для выравнивания полей (§21.4.5)
```

```
        basefield,   // флаги системы счисления целых (§21.4.2)
        floatfield,  // флаги вывода чисел с плав. запятой (§21.4.3)
        unitbuf;     // очистка буфера после каждой операции вывода

    fmtflags flags () const;                                   // читать флаги
    fmtflags flags (fmtflags f) ;                              // установить флаги

    fmtflags setf (fmtflags f) {return flags (flags () |f) ; }     // добавить флаг
    fmtflags setf (fmtflags f, fmtflags mask) {return flags ( (flags () &~mask) | (f&mask) ) ; }
    void unsetf (fmtflags mask) {flags (flags () &~mask) ; }     // очистить флаги
    // ...
};
```

Значения флагов зависят от реализации. Используйте только их символические имена, а не соответствующие им числовые значения, даже если вы уверены за эти значения для вашей конкретной реализации.

```
const ios_base::fmtflags my_opt = ios_base::left|ios_base::oct|ios_base::fixed;
```

Предоставление интерфейса управления в виде набора флагов вкупе с функциями для их установки и сброса хоть и выглядит несколько старомодно, зато проверено временем. Главное достоинство такого подхода заключается в том, что пользователь может легко комбинировать имеющиеся опции. Например:

```
void your_function (ios_base::fmtflags opt)
{
  ios_base::fmtflags old_options = cout.flags (opt) ;   // установить новые
                                                        // и сохранить старые опции
  // ...
  cout.flags (old_options) ;                            // восстановить старые опции
}

void my_function ()
{
  your_function (my_opt) ;
  // ...
}
```

Функция *flags*() возвращает старый набор флагов (опций).

Имея возможность читать и задавать любые опции, мы можем устанавливать значения отдельных флагов. Например:

```
myostream.flags (myostream.flags () |ios_base::showpos) ;
```

что заставляет поток myostream явно показывать знак + перед положительными числами, в то время как остальные опции остаются без изменений. Здесь мы прочли старый набор опций и с помощью логической операции «побитовое ИЛИ» внесли в него ***showpos***. Функция *setf*() делает то же самое, так что предыдущий пример с ее помощью записывается короче:

```
myostream.setf (ios_base::showpos) ;
```

После установки флаги сохраняют свое состояние до тех пор, пока не будут сброшены.

Управление опциями ввода/вывода путем явной установки и сброса флагов сложно, утомительно и чревато ошибками. В простых случаях лучше пользоваться

манипуляторами (*manipulators*) (§21.4.6). Управление состоянием потоков с помощью битовых флагов является неплохим методом изучения деталей реализации, но оно вряд ли пригодно для проектирования хорошего интерфейса.

21.4.1.1. Копирование состояния форматирования

Полное состояние форматирования потока можно скопировать функцией ***copyfmt***():

```
template<class Ch, class Tr = char_traits<Ch> >
class basic_ios: public ios_base
{
public:
  // ...
  basic_ios& copyfmt(const basic_ios& f);
  // ...
};
```

Буфер потока (§21.6) и состояние буфера функцией ***copyfmt***() не копируются. Тем не менее, все остальное копируется, в том числе запрашиваемые исключения (§21.3.6) и все пользовательские добавления к этому состоянию (§21.7.1).

21.4.2. Вывод целых

Рассмотренный выше прием с добавлением новых флагов при помощи функций ***flags***() или ***setf***() работает только тогда, когда единственный бит отвечает за ту или иную характеристику вывода. Это не проходит для системы счисления при выводе целых чисел или для стиля вывода чисел с плавающей запятой, так как для них невозможно установить отдельный бит или комбинацию битов.

С этой целью в ***<iostream>*** предоставляется версия функции ***setf***() с двумя аргументами, которой помимо устанавливаемого значения передается еще и второй аргумент, указывающий, что именно мы хотим установить. Например, следующий код

```
cout.setf(ios_base::oct,ios_base::basefield);
cout.setf(ios_base::dec,ios_base::basefield);
cout.setf(ios_base::hex,ios_base::basefield);
```

устанавливает необходимую систему счисления без каких-либо побочных воздействий на остальные части состояния форматирования потока. Будучи установленной, система счисления остается неизменной до тех пор, пока не будет переустановлена. Например:

```
cout<< 12334 << ' ' << 12334 << ' ';          // по умолчанию - десятичная
cout.setf(ios_base::oct,ios_base::basefield); // восьмеричная
cout<< 12334 << ' ' << 12334 << ' ';
cout.setf(ios_base::hex,ios_base::basefield); // шестнадцатеричная
cout<< 12334 << ' '<< 12334 << ' ';
```

выведет ***1234 1234 2322 2322 4d2 4d2***.

Если нужно явно указать систему счисления для каждого выводимого числа, то тогда следует указать ***showbase***. Таким образом, указав

```
cout.setf(ios_base::showbase);
```

до операций вывода, мы получим на выходе *1234 1234 02322 02322 0x4d2 0x4d2*. Стандартные манипуляторы предоставляют более изящный способ управления системой счисления для выводимых целых чисел (§21.4.6.2).

21.4.3. Вывод значений с плавающей запятой

Вывод чисел с плавающей запятой осуществляется в соответствии с их *форматом* и *точностью*:

- *Общий* формат делегирует реализации право выбрать наиболее подходящую форму отображения в соответствии с имеющейся для этого памятью. Точность определяет максимальное количество цифр. Это соответствует формату *%g* в функции *printf*() (§21.8).
- *Научный* формат представляет число одной цифрой до десятичной точки и экспонентой. Точность определяет количество цифр после десятичной точки. Это соответствует формату *%e* в функции *printf*().
- *Фиксированный* формат представляет число как целую часть, за которой следуют десятичная точка и дробная часть. Точность определяет максимальное количество цифр дробной части. Это соответствует формату *%f* в функции *printf*().

Мы управляем выводом чисел с плавающей запятой посредством функций, манипулирующих состоянием. В частности, мы можем задать форму вывода чисел с плавающей запятой, не оказывая влияния на иные аспекты состояния потока. Например:

```
cout << "default:\t" << 1234.56789 << '\n';

Cout.setf(ios_base::scientific,ios_base::floatfield);   // научный формат
cout << "scientific:\t" << 1234.56789 << '\n';

cout.setf(ios_base::fixed, ios_base::floatfield);   // формат с фиксированной точкой
cout << "fixed:\t" << 1234.56789 << '\n';

cout.setf(ios_base::fmtflags(0),ios_base::floatfield); // восстанав. формат по умолчанию
cout << "default:\t" << 1234.56789 << '\n';
```

порождает вывод

```
по умолчанию:     1234.57
научный:          1.234568e+03
фиксированный:    1234.567890
по умолчанию:     1234.57
```

Точность по умолчанию (для всех форматов) — 6 цифр. Точностью управляют функции-члены класса *ios_base*:

```
class ios_base
{
public:
  // ...
  streamsize precision() const;          // получить точность
  streamsize precision(streamsize n); // установка новой точности (возврат старой)
  // ...
};
```

Вызов функции ***precision***() оказывает влияние на все операции ввода/вывода в поток чисел с плавающей запятой, которое сохраняется вплоть до следующего вызова ***precision***(). Таким образом,

```
cout.precision(8);
cout << 1234.56789 << ' '<< 1234.56789 << ' ' << 123456 << '\n';
cout.precision(4)
cout << 1234.56789 << ' '<< 1234.56789 << ' ' << 123456 << '\n';
```

порождает вывод

```
1234.5679 1234.5679 123456
1235 1235 123456
```

Отметим, что числа с плавающей запятой округляются, а не урезаются, и еще отметим, что функция ***precision***() не влияет на вывод целых чисел.

Флаг ***uppercase*** (§21.4.1) определяет, что именно используется — ***e*** или ***E***, для обозначения экспоненты в научном формате.

Манипуляторы позволяют проще и изящнее задать формат вывода чисел с плавающей запятой (§21.4.6.2).

21.4.4. Поля вывода

Часто нужно заполнить текстом фиксированное пространство в выходной строке. Мы хотим использовать в точности ***n*** символов и не меньше (а больше — только если текст не помещается). Для этого мы определяем ширину поля и символ для его заполнения (при необходимости):

```
class ios_base
{
public:
  // ...
  streamsize width() const;                    // получить ширину поля
  streamsize width(streamsize wide);           // установить ширину поля
  // ...
};

template<class Ch, class Tr = char_traits<Ch> >
class basic_ios: public ios_base
{
public:
  // ...
  Ch fill() const;                             // получить символ-заполнитель
  Ch fill(Ch ch);                              // установить символ-заполнитель
  // ...
};
```

Функция ***width***() задает минимальное количество символов, которые будут выведены следующей операцией >> стандартной библиотеки для числовых значений, значений типа ***bool***, С-строк, символов, указателей (§21.2.1), строк типа ***string*** (§20.3.15) и контейнеров ***bitset*** (§17.5.3.3). Например:

```
cout.width(4);
cout << 12;
```

выведет *12* с двумя пробелами спереди.

Символ-заполнитель можно задать функцией *fill*(). Например, следующий код

```
cout.width(4);
cout.fill('#');
cout<< "ab";
```

выведет строку ##*ab*.

Умолчательным символом-заполнителем является пробел, а умолчательная ширина поля вывода равна нулю, что трактуется как «столько символов, сколько нужно». Умолчательное значение ширины поля вывода можно восстановить с помощью вызова

```
cout.width(0);
```

Вызов ***width*(*n*)** устанавливает минимальное число символов равным ***n***. При наличии большего числа символов все они будут выведены. Например:

```
cout.width(4);
cout << "abcdef";
```

выводит ***abcdef***, а не ***abcd***. Лучше получить некрасивый, но правильный вывод, чем прекрасный с виду, но неправильный (см. §21.10[21]).

Вызов ***width*(*n*)** влияет только на одну, непосредственно следующую операцию вывода <<:

```
cout.width(4);
cout.fill('#');
cout << 12 << ':' << 13;
```

Данный код выведет ##*12*:*13*, а не ##*12*###:##*13*, что имело бы место в случае, если вызов ***width*(*4*)** влиял бы на все последующие операции вывода. В последнем случае нам пришлось бы то и дело вызывать ***width***() практически для всех выводимых величин.

Стандартные манипуляторы позволяют задать ширину поля вывода проще и изящнее (§21.4.6.2).

21.4.5. Выравнивание полей

Управлять выравниванием символов в полях вывода можно вызовом функции *setf*():

```
cout.setf(ios_base::left,ios_base::adjustfield);
cout.setf(ios_base::right,ios_base::adjustfield);
cout.setf(ios_base::internal,ios_base::adjustfield);
```

Это позволяет задать выравнивание вывода в полях, ширина которых задается с помощью ***ios_base*::*width***(), без побочного влияния на остальные аспекты состояния потока.

Можно следующим образом задать необходимое выравнивание:

```
cout.fill('#');

cout<< '(';
cout.width(4);
cout<< -12 << "),(";
```

```
cout.width(4);
cout.self(ios_base::left,ios_base::adjustfield);
cout << -12 << "),(";

cout.width(4);
cout.setf(ios_base::internal,ios_base::adjustfield);
cout << -12 << ")";
```

Это породит вывод (#-*12*), (-*12*#), (-#*12*). Выравнивание ***internal*** располагает символ заполнитель между знаком и числовым значением. По умолчанию имеет место выравнивание влево. Результат не определен, если задано более одного флага выравнивания одновременно.

21.4.6. Манипуляторы

Чтобы избавить программиста от необходимости работать с состоянием потока на уровне флагов, стандартная библиотека предлагает набор функций для манипулирования состоянием. Ключевая идея — вставлять операции, изменяющие состояние потока, между последовательными операциями чтения или записи объектов. Например, мы можем явным образом сбрасывать выходной буфер:

```
cout << x <<flush << y << flush;
```

Здесь функция ***cout.flush***() вызывается в необходимом порядке. Для этого имеется соответствующая версия операции <<, которая принимает указатель на функцию и вызывает функцию по этому указателю:

```
template<ciass Ch, class Tr = char_traits<Ch> >
class basic_ostream: virtual public basic_ios<Ch,Tr>
{
public:
  // ...
  basic_ostream& operator<<(basic_ostream& (*f)(basic_ostream&)) {return f(*this);}
  basic_ostream& operator<<(ios_base& (*f)(ios_base&));
  basic_ostream& operator<<(basic_ios<Ch,Tr>& (*f)(basic_ios<Ch,Tr>&));
  // ...
};
```

Чтобы это работало, функция должна быть или статической функцией-членом, или глобальной функцией правильного типа. В частности, функция ***flush***() определяется следующим образом:

```
template<class Ch, class Tr = char traits<Ch> >
basic_ostream<Ch,Tr>& flush(basic_ostream<Ch,Tr>& s)
{
  return s.flush();       // вызов flush() - функции-члена ostream
}
```

Это объявление гарантирует, что выражение

```
cout << flush;
```

разрешается как вызов

```
cout.operator<<(flush);
```

приводящий к вызову

```
flush (cout) ;
```

который, в свою очередь, вызывает

```
cout.flush () ;
```

Все это делается (на стадии компиляции) ради того, чтобы вызов ***basic_ostream::flush()*** порождался из внешне простого выражения ***cout<<flush***.

Существует множество операций, которые нам хотелось бы выполнять непосредственно перед или сразу после операции ввода или вывода. Например:

```
cout << x;
cout.flush () ;
cout << y;
cin.unsetf(ios_base::skipws) ;     // не пропускать пробельные символы (§21.4.1)
cin >> x;
```

Когда операции разносятся по разным операторам, логические связи между ними становятся менее очевидными. А потеря логической ясности приводит к трудностям в понимании кода. Тут нам и приходят на помощь манипуляторы, которые позволяют вставлять операции типа ***flush()*** или ***unsetf(ios_base::skipws)*** в компактный список (последовательность) операций ввода и вывода. Например:

```
cout << x << flush << y;
cin >> noskipws >> x;
```

Заметьте, что манипуляторы определены в пространстве имен ***std***, так что их нужно в общем случае квалифицировать префиксом ***std::*** (если это не входит в текущую область видимости):

```
std::cout << endl;          // error: endl - вне области видимости
std::cout << std::endl;     // ok
```

Естественно, класс ***basic_istream*** предоставляет операции >>, способные активизировать манипуляторы так же, как и рассмотренный выше класс ***basic_ostream***:

```
template<class Ch, class Tr = char_traits<Ch> >
class basic_istream: virtual public basic_ios<Ch, Tr>
{
public:
  //...
  basic_istream& operator>> (basic_istream& (*pf) (basic_istream&)) ;
  basic_istream& operator>> (basic_ios<Ch, Tr>& (*pf) (basic_ios<Ch, Tr>&)) ;
  basic_istream& operator>> (ios_base& (*pf) (ios_base&)) ;
  // ...
};
```

21.4.6.1. Манипуляторы, принимающие аргументы

Манипуляторы, принимающие аргументы, также полезны. Например, мы могли бы написать:

```
cout << setprecision (4) << angle;
```

чтобы вывести значение переменной ***angle*** (имеющей тип чисел с плавающей запятой) из ***4*** цифр.

Для этого *setprecision*() возвращает объект, инициализированный числом *4*, и который при обращении к нему вызывает *cout.precision*(*4*). Такой манипулятор есть объект функционального класса, который активируется операцией <<, а не операцией (). Точный тип этого объекта зависит от реализации, но его можно определить следующим образом:

```
struct smanip
{
  ios_base& (*f) (ios_base&, int);   // функция для вызова
  int i;

  smanip(ios_base& (*ff) (ios_base&, int), int ii) : f(ff), i(ii) { }
};

template<class Ch, class Tr>
basic_ostream<Ch,Tr>& operatlor<< (basic_ostream<Ch,Tr>& os, const smanip& m)
{
  m.f(os,m.i);
  return os;

}
```

Конструктор класса *smanip* сохраняет свои аргументы в полях *f* и *i*, а *operator*<<() вызывает *f*(*i*). Теперь можно определить *setprecision*() следующим образом:

```
ios_base& set_precision(ios_base& s, int n)   // вспомогательная функция
{
  s.precision(n);                             // вызов функции-члена
  return s;
}

inline smanip setprecision (int n)
{
  return smanip(setprecision,n);              // создаем объект-функцию
}
```

и использовать его:

```
cout << setprecision(4) << angle;
```

При необходимости программист может определять новые манипуляторы в стиле *smanip* (§21.10[22]). При этом не требуется вносить изменения в определения шаблонов и классов стандартной библиотеки.

21.4.6.2. Стандартные манипуляторы ввода/вывода

Стандартная библиотека предоставляет манипуляторы, соответствующие различным состояниям форматирования и их изменениям. Стандартные манипуляторы определены в пространстве имен *std*. Манипуляторы, использующие *ios_base*, представлены в заголовочном файле <*ios*>. Манипуляторы, использующие *istream* и *ostream*, представлены в <*istream*> и <*ostream*>, соответственно, и, кроме того, в <*iostream*>. Остальные стандартные манипуляторы представлены в <*iomanip*>.

```
ios_base& boolalpha(ios_base&);        // символьное представление true и false
ios_base& noboolalpha(ios_base& s);    // s.unsetf(ios_base::boolalpha)
```

```
ios_base& showbase(iosjfrase&);              // on output prefix oct by 0 and hex by Ox
ios_base& noshowbase(ios_base& s);           // s.unsetf(ios_base::showbase)

ios_base& showpoint(ios_base&;
ios_base& noshowpoint(ios_base& s);          // s.unsetf(ios_base::showpoint)

ios_base& showpos(ios_base&);
ios_base& noshowpos(ios_base& s);            // s.unsetf(ios_base::showpos)

ios_base& skipws(ios_base&);                 // пропускать пробельные символы
ios_base& noskipws(ios_base& s);             // s.unsetf(ios_base::skipws)

ios_base& uppercase(ios_base&);              // X и E, а не x и e
ios_base& nouppercase(ios_base&);            // x и e, а не X и E

ios_base& internal(ios_base&);               // выравнивание §21.4.5
ios_base& left(ios_base&);                   // заполнитель после значение
ios_base& right(ios_base&);                  // заполнитель перед значением

ios_base& dec(ios_base&);                    // целые по основанию 10 (§21.4.2)
ios_base& hex(ios_base&);                    // по основанию 16
ios_base& oct(ios_base&);                    // по основанию 8

// формат чисел с плавающей запятой:
ios_base& fixed(ios_base&);                  // формат dddd.dd (§21.4.3)
ios_base& scientific(ios_base&);             // формат d.ddddEdd

template<class Ch, class Tr>
basic_ostream<Ch,Tr>& endl(basic_ostream<Ch,Tr>&); // поместить '\n' и flush
template<class Ch, class Tr>
basic_ostream<Ch,Tr>& ends(basic_ostream<Ch,Tr>&); // поместить '\0'
template<class Ch, class Tr>
basic_ostream<Ch,Tr>& flush(basic_ostream<Ch,Tr>>&);// очистить буфер потока
template<class Ch, class Tr>
basic_istream<Ch,Tr>& ws(basic_istream<Ch,Tr>&);    // съесть пробельные символы

smanip resetiosflags(ios_base::fmtflags f);  // очистить флаги (§21.4)
smanip setiosflags(ios_base::fmtflags f);    // установить флаги (§21.4)
smanip setbase(int b);                       // вывод целых по основанию b (§21.4.2)
smanip setfill(int c);                       // сделать с символом-заполнителем (§21.4.4)
smanip setprecision(int n);                  // n цифр (§21.4.3, §21.4.6)
smanip setw(int n);                          // ширина след. поля - n символов (§21.4.4)
```

Например,

```
cout << 1234 << ',' << hex << 1234 << ',' << oct << 1234 << endl;
```

выведет 1234, 4d2, 2322, а

```
cout<< '(' << setw(4) << setfill('#')<< 12 <<") (" << 12 << ")\n";
```

выведет (##12) (12).

Применяя манипуляторы без аргументов, *не ставьте круглых скобок*. Используя стандартные манипуляторы с аргументами, не забудьте вставить в исходный код директиву препроцессора #*include* <*iomanip*>. Например:

```
#include <iostream>
using namespace std;

int main ()
{
  cout << setprecision (4)  // error: setprecision не определен (забыли <iomanip>)
      << scientific ()       // error: ostream<<ostream& (неуместные скобки)
      << 1.41421 << endl;
}
```

21.4.6.3. Манипуляторы, определяемые пользователем

Программист может создавать манипуляторы в стиле стандартных манипуляторов. Здесь я представлю иной (дополнительный) стиль, который может оказаться полезным при форматировании чисел с плавающей запятой.

Как мы знаем, установка точности вызовом ***precision***() сохраняется для всех операций вывода, а ***width***() применяется лишь к ближайшей операции. Мне же хочется определить нечто такое, что упростит вывод чисел с плавающей запятой в предопределенном формате без влияния на последующие операции вывода в поток. Для этого я определю класс, представляющий форматы, а также класс, который представляет формат плюс форматируемое значение, и, наконец, функцию ***operator***<<() (то есть операцию <<), которая выводит значение в соответствии с форматом. Например:

```
Form gen4 (4) ;                    // общий формат, точность - 4

void f(double d)
{
  Form sci8 = gen4;
  sci8.scientific () .precision (8) ;    // научный формат, точность - 8
  cout << d << ' ' <<gen4 (d) << ' ' << sci8(d) << ' ' << d << '\n';
}
```

Вызов *f*(*1234.56789*) выведет

```
1234.57 1235 1.23456789e+03 1234.57
```

Заметьте, что использование Form не влияет на состояние потока, так что последний вывод числа ***d*** имеет тот же умолчательный формат, что и первый.

Вот упрощенная реализация:

```
class Bound_form;

class Form
{
  friend ostream& operator<< (ostream&, const Bound_form&) ;

  int prc;                         // точность
  int wdt;                         // ширина
  int fmt;                         // общий, научный, фиксированный (§21.4.3)
  // ...

public:
  explicit Form (int p = 6) : prc (p)   // умолчательная точность - 6
  {
    fmt = 0;                       // общий формат (§21.4.3)
    wdt = 0;                       // ширина по необходимости
  }
```

```
    Bound_form operator() (double d) const;

    Form& scientific() {fmt = ios_base::scientific; return* this; }
    Form& fixed() {fmt = ios_base::fixed; return *this; }
    Form& general() {fmt = 0; return *this; }

    Form& uppercase();
    Form& lowercase();
    Form& precision(int p) {prc = p; return *this; }

    Form& width(int w) {wdt = w; return *this; }  // применяется ко всем типам
    Form& fill(char);

    Form& plus(bool b = true);                   // явный плюс
    Form& trailing_zeros(bool b = true);         // выводить хвостовые нули
    // ...
};
```

Идея заключается в том, что ***Form*** содержит всю информацию, необходимую для форматирования одного элемента данных. Выбранные умолчательные значения разумны во многих случаях, а разные функции-члены можно использовать для установки отдельных аспектов форматирования. Операция `()` используется для связывания значения с форматом, выбранным для его вывода. Тогда ***Bound_form*** можно вывести в заданный поток подходящей функцией ***operator***<<`()` (операцией <<):

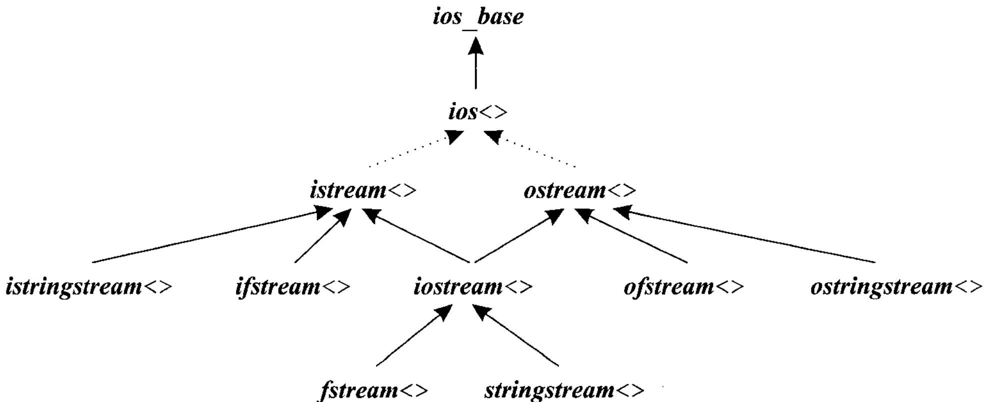
```
struct Bound_form
{
    const Form& f;
    double val;

    Bound_form(const Form& ff, double v) : f(ff), val(v) { }
};

Bound_form Form::operator() (double d) const { return Bound_form(*this, d); }

ostream& operator<<(ostream& os, const Bound_form& bf)
{
    ostringstream s;           // строковые потоки описаны в §21.5.3
    s.precision(bf.f.prc);
    s.setf(bf.f.fmt, ios_base::floatfield);
    s.width(bf.f.wdt);
    s << bf.val;               // составляем строку в s
    return os << s.str();      // выводим s в os
}
```

Написание менее примитивной реализации операции << я оставляю в качестве упражнения (§21.10[21]). Классы ***Form*** и ***Bound_form*** легко расширяются для форматирования целых чисел, строк типа ***string*** и т.д. (см. §21.10[20]).

Отметим, что представленные объявления превращают комбинацию операций << и `()` в тернарную операцию; ***cout***<<***sci4***(***d***) собирает ***ostream***, формат и значение в единую функцию до того, как выполнить какое-либо реальное вычисление.

21.5. Файловые и строковые потоки

Когда программа на C++ запускается на выполнение, потоки ***cout***, ***cerr***, ***clog***, ***cin*** и их «широкие» аналоги (§21.2.1) уже готовы к применению. Они доступны по умолчанию, а их связь с устройствами ввода/вывода определяется «системой». Дополнительно, вы можете создать собственные потоки. В этом случае вы должны указать, к чему эти потоки нужно прикрепить. Прикрепления к файлу или к строке string весьма типичны, так что они непосредственно поддерживаются стандартной библиотекой. Рассмотрим иерархию стандартных потоковых классов:



Классы, оканчивающиеся на <> — это шаблоны, параметризуемые типом символов, а их имена имеют префикс ***basic_***. Пунктирная линия соответствует виртуальному базовому классу (§15.2.4).

Файлы и строки являются примерами контейнеров, куда вы можете писать и откуда вы можете считывать данные. Следовательно, вы можете завести поток, поддерживающий как операцию <<, так и операцию >>. Такой поток называется ***iostream***, он определен в пространстве имен ***std*** и представлен в заголовочном файле <***iostream***>:

```
template<class Ch, class Tr = char_traits<Ch> >
class basic_iostream : public basic_istream<Ch, Tr>, public basic_ostream<Ch, Tr>
{
public:
  explicit basic_iostream (basic_streambuf<Ch, Tr>* sb) ;
  virtual ~basic_iostream () ;
};

typedef basic_iostream<char> iostream;
typedef basic_iostream<wchar_t> wiostream;
```

Чтение из и запись в ***iostream*** контролируется операциями «взять из буфера» и «положить в буфер», выполняемыми над буфером ***streambuf*** потока ***iostream*** (§21.6.4).

21.5.1. Файловые потоки

Рассмотрим программу, которая копирует один файл в другой. Имена файлов задаются аргументами командной строки:

```
#include <fstream>                    // файловые потоки и операции
#include <iostream>                   // cin, cout, cerr, и т.д.
#include <cstdlib>                    // exit(), и т.д.

void error (const char* p, const char* p2 = "")
{
  std::cerr << p << ' ' << p2 << '\n';
  std::exit (1) ;
}

int main (int argc, char* argv[ ])
{
  if(argc != 3) error ("wrong number of arguments") ;

  std::ifstream from (argv[1]) ;          // открываем файловый поток на вход
  if(!from) error ("cannot open input file", argv[1]) ;

  std::ofstream to (argv[2]) ;            // открываем файловый поток на выход
  if(!to) error ("cannot open output file", argv[2]) ;

  char ch;
  while (from.get(ch)) to.put(ch) ;       // копируем символы

  if(!from.eof() || !to) error ("something strange happened") ;
}
```

Файл открывается для ввода созданием объекта класса ***ifstream*** (входной файловый поток) с именем класса в качестве аргумента. Подобным же образом файл для вывода открывается созданием объекта класса ***ofstream*** (выходной файловый поток) с именем класса в качестве аргумента. В обоих случаях, мы проверяем состояние созданных объектов, чтобы удостовериться в успешном открытии файлов.

Класс ***basic_ofstream*** определяется в заголовочном файле ***<fstream>***:

```
template<class Ch, class Tr = char_traits<Ch> >
class basic_ofstream: public basic_ostream<Ch, Tr>
{
public:
  basic_ofstream () ;
  explicit basic_ofstream (const char* p, openmode m = out) ;

  basic_filebuf<Ch, Tr>* rdbuf() const;    // получить указатель на текущий
                                            // файловый буфер (§21.6.4)

  bool is_open () const;
  void open (const char* p, openmode m = out) ;
  void close () ;
};
```

Класс ***basic_ifstream*** похож на ***basic_ofstream***, за исключением того, что является производным от ***basic_istream*** и по умолчанию открыт для чтения. Дополнительно, стандартная библиотека предлагает класс ***basic_fstream***, который похож на ***basic_of-***

stream за исключением того, что он наследуется от ***basic_iostream*** и по умолчанию открыт как на чтение, так и на запись.

Как обычно, с помощью операторов ***typedef*** определяются наиболее распространенные типы:

```
typedef basic_ifstream<char> ifstream;
typedef basic_ofstream<char> ofstream;
typedef basic_fstream<char>  fstream;

typedef basic_ifstream<wchar_t> wifstream;
typedef basic_ofstream<wchar_t> wofstream;
typedef basic_fstream<wchar_t>  wfstream;
```

Второй аргумент конструкторов классовых потоков определяет альтернативные режимы открытия:

```
class ios_base
{
public:
  // ...
  typedef implementation_defined3 openmode;

  static openmode app,       // добавление в конец
                  ate,       // открыть и перейти к концу файла
                  binary,    // I/O в бинарном режиме (а не текстовом)
                  in,        // открыть на чтение
                  out,       // открыть на запись
                  trunc;     // урезать файл до нулевой длины
  // ...
};
```

Действительные значения для полей типа ***openmode*** и их смысл зависят от реализации. Проконсультируйтесь со справочником по вашей реализации и поэкспериментируйте. Комментарии также помогают определить основное назначение режимов. Например, мы можем открыть файл для добавления данных в конец файла:

```
ofstream mystream (name.c_str(), iosbase::app);
```

Также можно открыть файл для чтения и записи. Например:

```
fstream dictionary ("concordance", ios_base::in | ios_base::out);
```

21.5.2. Закрытие потоков

Файл можно явным образом закрыть, вызвав функцию ***close***() для его потока:

```
void f(ostream& mystream)
{
  // ...
  mystream.close();
}
```

Неявным образом это делается деструктором потока. Поэтому явный вызов функции ***close***() нужен лишь тогда, когда мы хотим закрыть файл до выхода из области видимости, в которой поток объявлен.

В связи с этим возникает вопрос, как реализации могут гарантировать создание предопределенных потоков ***cout***, ***cin***, ***cerr*** и ***clog*** до их первого использования и закрытие этих потоков строго после их последнего использования? Ясно, что разные реализации <***iostream***> с этой целью могут использовать разные приемы. Более того, детали реализации должны быть скрыты от пользователя. Здесь я просто приведу один достаточно общий прием, гарантирующий правильный порядок создания и уничтожения глобальных объектов различных типов. Коммерческие реализации могут эту задачу решить эффективнее за счет применения специфических черт конкретных компиляторов и компоновщиков.

Основная идея состоит в том, чтобы определить вспомогательный класс, подсчитывающий, сколько раз заголовочный файл <***iostream***> был включен в отдельно компилируемые исходные файлы:

```
class ios_base::Init {
  static int count;

public:
  Init();
  ~Init();
};

namespace {ios_base::Init __ioinit; }     // в <iostream>

int cos_base::Init::count = 0;            // в некоторый .c-файл
```

Каждая единица трансляции (§9.1) объявляет свой собственный объект с именем ***__ioinit***. Конструктор объекта ***__ioinit*** использует ***ios_base::Init::count*** как флаг первого вызова, чтобы гарантировать однократную инициализацию глобальных объектов потоковой библиотеки ввода/вывода:

```
ios_base::Init::Init() (if(count++ = 0) { /*инициализация cout,cerr,cin,etc.*/ }
```

Аналогично, деструктор объекта ***__ioinit*** использует ***ios_base::Init::count*** как флаг последнего вызова, чтобы гарантировать закрытие потоков:

```
ios_base::Init::~Init()
{
  if(--count == 0) { /*очистка cout(flush,и т.д.),cerr,cin и т.д.*/ }
}
```

Это общая техника программирования для работы с библиотеками, требующими инициализации и очистки глобальных объектов. В системах, где весь код во время исполнения находится в оперативной памяти, рассмотренный прием практически не имеет ограничений. В противном случае загрузка всех объектных файлов в память для выполнения их инициализирующих функций может оказаться слишком накладной. В общем случае лучше избегать глобальных объектов. Для класса, где каждая операция выполняет значительный объем работы, лучше именно в них тестировать флаг первого вызова (вроде ***ios_base::Init::count***) для гарантирования инициализации. В то же время для потоков такой подход слишком дорог — суммарные накладные расходы на тестирование флага первого вызова в каждой операции чтения/записи одного символа будут чрезмерно велики.

21.5.3. Строковые потоки

Поток можно прикрепить к строке типа ***string***. То есть мы можем читать и писать в строку при помощи средств форматирования потоков. Такие потоки называют (обобщенно) ***stringstream***. Они определяются в <***sstream***>:

```
template<class Ch, class Tr = char_traits<Ch>, class A = allocator<Ch> >
class basic_stringstream: public basic_iostream<Ch,Tr>
{
public:
  explicit basic_stringstream(ios_base::openmode m = out|in);
  explicit basic_stringstream(const basic_string<Ch,Tr,A>& s, openmode m = out|in);

  basic_string<Ch,Tr,A> str() const;          // получить копию строки
  void str(const basic_string<Ch,Tr,A>& s);   // установить значение копии s

  basic_stringbuf<Ch,Tr,A>* rdbuf() const;    // получить указатель на текущий
                                              // файловый буфер
};
```

Класс ***basic_istringstream*** похож на ***basic_stringstream*** за исключением того, что он наследуется от ***basic_istream*** и по умолчанию открыт на чтение. Класс ***basic_ostringstream*** похож на ***basic_stringstream*** за исключением того, что он наследуется от ***basic_ostream*** и по умолчанию открыт на запись.

Как обычно, с помощью операторов ***typedef*** определяются наиболее распространенные специализации:

```
typedef basic_istringstream<char> istringstream;
typedef basic_ostringstream<char> ostringstream;
typedef basic_stringstream<char>  stringstream;

typedef basic_istringstream<wchar_t> wistringstream;
typedef basic_ostringstream<wchar_t> wostringstream;
typedef basic_stringstream<wchar_t>  wstringstream;
```

Например, можно применить тип ***ostringstream*** для форматирования строковых сообщений:

```
string compose(int n, const string& cs)
{
  extern const char* std_message[];
  ostringstream ost;
  ost<< "error(" << n << ") " << std_message[n] << " (user comment: " << cs << ')';
  return ost.str();
}
```

Нет необходимости следить за переполнением, поскольку ***ost*** обеспечивает автоматическое расширение внутреннего хранилища. Этот прием может оказаться особо полезным в случаях копирования, когда требуется более сложное форматирование, чем то, что используется при выводе на строкоориентированные устройства.

Начальное значение для строкового потока задается аналогично тому, как файл задается для файлового потока:

```
string compose2(int n, const string& cs)
{
  extern const char* std_nessage[];
```

```
    ostringstream ost ("error (", ios_base::ate) ;
    ost << n << ") " << std_message[n] << " (user comment: " << cs << ')';
    return ost.str () ;
}
```

Входной поток *istringstream* читает данные из своей инициализирующей строки (как поток *ifstream* читает из файла, для которого он инициализирован):

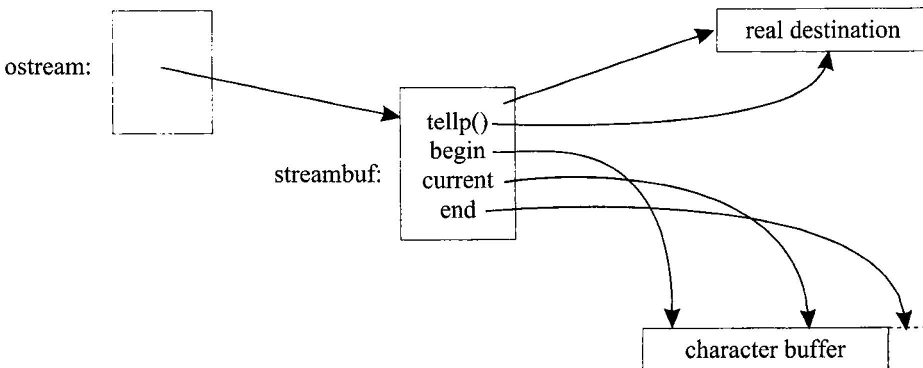
```
#include <sstream>

void word_per_line (const string& s)        // по одному слову в одну строку
{
    istringstream ist (s) ;
    string w;
    while (ist>>w) cout << w << '\n';
}

int main ()
{
    word_per_line ("If you think C++ is difficult, try English") ;
}
```

Инициализирующая строка копируется в *istringstream*. Завершение чтения отдельных слов из *ist* наступает по исчерпанию содержимого этой строки.

Можно определить потоки, которые осуществляют чтение и запись в массивы символов (§21.10[26]). Это бывает полезно при работе со старыми программами, тем более, что предназначенные для этого классы *ostrstream* и *istrstream* с самого начала входили в библиотеку потоков.

21.6. Буферирование

Вообще говоря, выходной поток кладет символы в буфер. Через какое-то время они попадают туда, куда нужно. Такой буфер называется *streambuf* (§21.6.4), и определен он в заголовочном файле <*streambuf*>. Различные реализации *streambuf* используют разные стратегии буферизации. В типичном случае *streambuf* хранит символы в массиве до тех пор, пока его переполнение не заставит вывести символы по назначению. В итоге, работа ostream может быть графически отображена следующим образом:



Шаблонные аргументы для ***ostream*** и ***streambuf*** должны совпадать и определять тип символов.

Поток ***istream*** вполне аналогичен, просто символы передаются в ином направлении.

Небуферированный ввод/вывод — это такой же ввод/вывод, только ***streambuf*** немедленно передает каждый символ по назначению, а не ждет, когда накопится нужное число символов, оптимальное для дальнейшей передачи.

21.6.1. Потоки вывода и буферы

Класс ***ostream*** определяет операции для преобразования значений разных типов в последовательности символов согласно принятым соглашениям (§21.2.1) и явным директивам форматирования (§21.4). Кроме того, ***ostream*** предоставляет операции, напрямую работающие со ***streambuf***:

```
template<class Ch, class Tr = char_traits<Ch> >
class basic_ostream: virtual public basic_ios<Ch,Tr>
{
public:
  //...
  explicit basic_ostream(basic_streambuf<Ch,Tr>* b);

  pos_type tellp();                                   // получить текущую позицию
  basic_ostream& seekp(pos_type);                     // установить текущую позицию
  basic_ostream& seekp(off_type, ios_base::seekdir);  // установить текущую позицию

  basic_ostream& flush();                             // очистить буфер (real destination)

  basic_ostream& operator<<(basic_streambuf<Ch,Tr>* b); // писать из b
};
```

Конструктор класса ***ostream*** принимает ***streambuf***, который и определяет, как именно обрабатываются символы и куда они в конце концов уходят. Например, ***ostringstream*** (§21.5.3) или ***ofstream*** (§21.5.1) создаются посредством инициализации потока ***ostream*** подходящим ***streambuf*** (§21.6.4).

Функции ***seekp***() используются для установки в ***ostream*** позиции записи. Суффикс ***p*** здесь как раз и означает позицию для записи (вставки — putting) символа в поток. Эти функции не оказывают никакого эффекта до тех пор, пока поток не прикреплен к чему-либо, для чего позиционирование имеет смысл, например к файлу. Тип ***pos_type*** соответствует позициям в файле, а тип ***off_type*** характеризует смещения от точки, индицируемой с помощью ***ios_base::seekdir***:

```
class ios_base
{
  //...
  typedef implementation_defined4 seekdir;
  static const seekdir beg,    // от начала текущего файла
                       cur,    // от текущей позиции
                       end;    // с конца файла в обратном направлении
  //...
};
```

Позиции в потоке нумеруются, начиная с *нуля*, так что мы можем думать о файле как о массиве из ***n*** символов. Например:

```
int f(ofstream& fout)          // fout ссылается на некоторый файл
{
  fout.seekp(10);
  fout << '#';                 // добавляем символ и сдвигаем позицию (+1)
  fout.seekp(-2, ios_base::cur);
  fout << '*';
}
```

Этот код поместит символ # в *file*[*10*], а символ * — в *file*[*9*]. Ничего подобного вроде произвольного доступа к элементам «простых» потоков ***istream*** или ***ostream*** нет (см. §21.10[13]). Попытки установить позицию чтения вне действительного интервала элементов (до файла или за его конец) переводят поток в состояние ***bad***() (§21.3.3).

Операция ***flush***() позволяет очистить буфер, не дожидаясь его переполнения.

Можно применить операцию << для непосредственного сброса содержимого ***streambuf*** в ***ostream***. Это полезно тем, кто реализует механизмы ввода/вывода.

21.6.2. Потоки ввода и буферы

Класс ***istream*** определяет операции для чтения символов и преобразования их в значения различных типов (§21.3.1). Кроме того, ***istream*** предоставляет операции, напрямую работающими со ***streambuf***.

```
template<class Ch, class Tr = char_traits<Ch> >
class basic_istream: virtual public basic_ios<Ch,Tr>
{
public:
  // ...
  explicit basic_istream(basic_streambuf<Ch,Tr>* b);

  pos_type tellg();                                    // получить текущую позицию
  basic_istream& seekg(pos_type);                      // установить текущую позицию
  basic_istream& seekg(off_type, ios_base::seekdir);  // установить текущую позицию

  basic_istream& putback(Ch c);    // поместить c обратно в буфер
  basic_istream& unget();          // возвратить последний считанный символ
  int_type peek();                 // взглянуть на символ, подлежащий чтению

  int sync();                      // очистить буфер (flush)

  basic_istream& operator>>(basic_streambuf<Ch,Tr>* b);   // читать в b
  basic_istream& get(basic_streambuf<Ch,Tr>>& b, Ch t = Tr::newline());

  streamsize readsome(Ch* p, streamsize n);           // читать не более n симв.
};
```

Функции позиционирования для ***istream*** работают точно так же, как их аналоги для ***ostream*** (§21.6.1). Суффикс ***g*** здесь как раз и означает позицию для чтения (***getting***) символа из потока. Суффиксы ***p*** и ***g*** нужны потому, что мы можем создать поток класса ***iostream***, производного одновременно от ***istream*** и ***ostream***, и такой поток должен будет отслеживать как позицию для записи, так и позицию для чтения.

Функция ***putback***() позволяет программе положить символ в поток ***istream***, чтобы он стал в нем «следующим символом для чтения», как это показано в §21.3.5. Функция ***unget***() возвращает назад самый последний из прочитанных из потока

символов. К сожалению, возврат назад прочитанных из входного потока символов не всегда возможен. Например, попытка вернуть назад прочитанный символ в начале буфера потока установит *ios_base::failbit*. К тому же, гарантируется возможность возврата только одного символа после успешного чтения. Функция *peek*() читает символ и оставляет его в *streambuf* так, что его можно прочитать снова. Таким образом, *c=peek*() эквивалентно (*c=get*(), *unget*(), *c*). Заметьте, что установка *failbit* может сгенерировать исключение (§21.3.6).

Сброс потока *istream* (синхронизация ввода с буфером) выполняется функцией *sync*(). Это не всегда можно выполнить корректно. Для некоторых видов потоков нам пришлось бы при этом повторно считать символы с реального источника — это и не всегда возможно, и не всегда желательно. Функция *sync*() при успешном выполнении возвращает *0*, а в противном случае возвращает *-1* и устанавливает *ios_base::badbit* (§21.3.3). Снова напомним, что установка *badbit* может вызвать исключение (§21.3.6). Функция *sync*() для буфера, связанного с потоком вывода, сбрасывает содержимое буфера на устройство вывода.

Операции >> и *get*(), нацеленные на чтение из *streambuf*, интересны в первую очередь разработчикам средств ввода/вывода. Только разработчикам следует напрямую манипулировать буферами *streambuf*.

Функция *readsome*() реализует низкоуровневую операцию, позволяющую пользователю «заглянуть в поток» и посмотреть, есть ли в нем символы, доступные для чтения. Это может быть полезно тогда, когда нежелательно дожидаться ввода, например с клавиатуры. См. также функцию *in_avail*() (§21.6.4).

21.6.3. Потоки и буферы

Связь между потоком и его буфером поддерживается классом *basic_ios*:

```
template<class Ch, class Tr = char_traits<Ch> >
class basic_ios : public ios_base
{
public:
  // ...
  basic_streambuf<Ch, Tr>* rdbuf() const;  // получить буфер

  // установить буфер, очистить, и вернуть указатель на старый буфер:
  basic_streambuf<Ch, Tr>* rdbuf(basic_streambuf<Ch, Tr>* b);

  locale imbue(const locale& loc);         // установить локализацию (и вернуть старую)

  char narrow(char_type c, char d) const;  // получить char из char_type
  char_type widen(char c) const;           // получить char_type из char
  // ...

protected:
  basic_ios();
  void init(basic_streambuf<Ch, Tr>* b);   // задать начальный буфер
};
```

Кроме чтения и установки *streambuf* для потока (§21.6.4), класс *basic_ios* предоставляет функцию *imbue*(), чтобы читать или фиксировать национальные черты (локализация, интернационализация) потока (§21.7) посредством вызова *imbue*() для его *ios_base* (§21.7.1) и вызова *pubimbue*() для буфера потока (§21.6.4).

Функции ***narrow***() и ***widen***() используются для преобразования символов типа ***char*** к символьному типу буфера ***char_type*** и обратно. Вторым аргументом вызова ***narrow***(***c***, ***d***) служит символ типа ***char***, возвращаемый функцией в том случае, когда невозможно преобразовать символ с типа ***char_type*** к типу ***char***.

21.6.4. Буфера потоков

Операции ввода/вывода специфицируются безо всякой связи с типами файлов, однако не со всеми устройствами можно обращаться одинаково в смысле буферизации. Например, поток ***ostream***, связанный со строкой типа ***string*** (§21.5.3), нуждается в буфере иного типа, чем ***ostream***, связанный с файлом (§21.5.1). Эта проблема решается предоставлением буферов разных типов во время инициализации потоков разных типов. Для работы со всеми типами буферов предоставляется единственный набор функций, так что ***ostream*** не содержит кода, позволяющего различить эти типы. Различные типы буферов являются производными от класса ***streambuf***, который предоставляет виртуальные функции для выполнения операций с разными стратегиями, например, функции для обработки переполнения и исчерпания.

Класс ***basic_streambuf*** задает два интерфейса. Открытый интерфейс предназначен в первую очередь для программистов, реализующих потоки вроде ***istream***, ***ostream***, ***fstream***, ***stringstream*** и т.д. Кроме того, предоставляется защищенный интерфейс для тех, кто реализует новые стратегии буферизации и классы типа ***sreambuf*** для новых типов источников ввода и устройств вывода.

Для понимания ***streambuf*** будет полезно сначала рассмотреть лежащую в его основе концепцию буферной области, доступ к которой осуществляется через защищенный интерфейс. Представим, что ***streambuf*** располагает *областью записи* (*put area*) и *областью чтения* (*get area*), с которыми работают операции << и >>, соответственно. Каждая их этих областей характеризуется указателем на начало, текущим указателем и указателем на элемент, расположенный за концевым элементом (one-past-the-end pointer). Эти указатели доступны через следующие функции:

```
template<class Ch, class Tr = char_traits<Ch> >
class basic_streambuf
{
protected:
  Ch* eback() const;                              // начало get-буфера
  Ch* gptr() const;                               // следующий заносимый символ
  Ch* egptr() const;                              // one-past-end для get-буфера

  void gbump(int n);                              // добавить n к gptr()
  void setg(Ch* begin, Ch* next, Ch* end);        // задать eback(), gptr() и egptr()

  Ch* pbase() const;                              // начало put-буфера
  Ch* pptr() const;                               // следующий свободный char
  Ch* epptr() const;                              // one-past-end для put-буфера
  void pbump(int n);                              // добавить n к pptr()
  void setp(Ch* begin, Ch* end);                  // установить pbase() и pptr() на начало, а
                                                  // epptr() - на конец

  // ...
};
```

Располагая массивом символов, функции *setp*() и *setg*() могут установить значения указателей. Реализация может обратиться к области чтения (get area) следующим образом:

```
template<class Ch, class Tr = char_traits<Ch> >
basic_streambuf<Ch,Tr>::int_type basic_streambuf<Ch,Tr>::snextc()
// пропустить текущий символ и прочитать последующий
{
  if(1 < egptr()-gptr())                  // если в буфере не меньше двух символов
  {
    gbump(1);                             // пропустить текущий символ
    return Tr::to_int_type(*gptr());      // вернуть новый текущий символ
  }

  if(1 == egptr()-gptr())                 // если в буфере ровно один символ
  {
    gbump(1);                             // пропустить текущий символ
    return underflow();
  }

  // буфер пуст (или отсутствует), пробуем заполнить:
  if(Tr::eq_int_type(uflow(),Tr::eof())) return Tr::eof();
  if(0< egptr()-gptr()) return Tr::to_int_type{*gptr());
  return underflow();
}
```

Доступ к буферу осуществляется через *gptr*(); функция *egptr*() маркирует границу области чтения. Символы читаются из реального источника функциями *uflow*() и *underflow*(). Вызовы *traits_type::to_int_type*() гарантируют, что этот код не зависит от истинного типа символов. Представленный код допускает множество типов потоковых буферов и учитывает, что виртуальные функции *uflow*() и *underflow*() могут перейти к новой области чтения (с помощью *setg*()).

Открытый интерфейс *streambuf* выглядит следующим образом:

```
template< class Ch, class Tr = char_traits<Ch> >
class basic_streambuf
{
public:
  // обычные typedef (§21.2.1)

  virtual ~basic_streambuf();

  locale pubimbue(const locale& loc);    // установить локализацию (вернуть старую)
  locale getloc() const;                  // получить локализацию

  basic_streambuf* pubsetbuf(Ch* p, streamsize n);         // задать размер буфера

  pos_type pubseekoff(off_type off, ios_base::seekdir way,   // позиция (§21.6.1)
                 ios_base::openmode m = ios_base::in|ios_base::out);
  pos_type pubseekpos(pos_type p,
                ios_base::openmode m = ios_base::in|ios_base::out);

  int pubsync();                          // ввод sync() (§21.6.2)

  int_type snextc();                      // пропустить текущ. символ, взять следующий
  int_type sbumpc();                      // продвинуть gptr() на 1
```

```
    int_type sgetc () ;                        // получить текущий символ
    streamsize sgetn (Ch* p, streamsize n) ;   // прочитать в p[0]..p[n-l]

    int_type sputbackc (Ch c) ;                // поместить с обратно в буфер (§21.6.2)
    int_type sungetc () ;                      // отменить чтение последнего символа

    int_type sputc (Ch c) ;                          // put c
    streamsize sputn (const Ch* p, streamsize n) ;   // записать в p[0]..p[n-l]
    streamsize in_avail () ;                         // ввод готов?
    // ...
};
```

Открытый интерфейс содержит функции для вставки символов в буфер и извлечения их из буфера. Код этих функций прост и легко допускает встраивание, а это важно для эффективности.

Функции, которые реализуют часть специфической стратегии буферизации, вызывают соответствующие функции защищенного интерфейса. Например, ***pubsetbuf***() вызывает функцию ***setbuf***(), которая замещается в производных классах с целью реализации специфической стратегии этого класса в получении памяти для буферизации символов. Применение двух функций для реализации операции вроде ***setbuf***() позволяет разработчику ***ostream*** выполнить некоторые вспомогательные действия до и после пользовательского кода. Например, разработчик может «обернуть» вызов виртуальной функции ***try***-блоком и перехватить исключения, возникающие в пользовательском коде.

По умолчанию, ***setbuf***(***0***, ***0***) означает отсутствие буферизации, a ***setbuf***(***p***, ***n***) означает применение массива ***p***[***0***]..***p***[***n-1***] для хранения буферизуемых символов.

Вызов ***in_avail***() используется для того, чтобы узнать, сколько символов находится в буфере. Это может потребоваться для того, чтобы не ждать ввода. При считывании из потока, связанного с клавиатурой, ***cin.get***(***c***) может ожидать пользователя, пока тот не вернется с обеда. На некоторых системах и в некоторых приложениях разумно учесть это обстоятельство при чтении. Например:

```
if(cin.rdbuf()->in_avail())     // get() не заблокирует
{
  cin.get(c);
  // что-нибудь сделать
}
else                            // get() может заблокировать
{
  // сделать что-нибудь еще
}
```

Отметим, что на некоторых системах бывает трудно определить, доступно ли что-нибудь для ввода. Таким образом, реализация (неудачная) функции ***in_avail***() могла бы возвращать ***нуль*** в случаях, когда предполагается успешность операции ввода.

В дополнение к открытому интерфейсу, который используется классами ***basic_istream*** и ***basic_ostream***, класс ***basic_streambuf*** предлагает защищенный интерфейс для разработчиков потоковых буферов. Именно тут и объявляются виртуальные функции, определяющие политику буферизации:

```
template<class Ch, class Tr = char_traits<Ch> >
class basic_streambuf
{
protected:
  // ...
  basic_streambuf() ;

  virtual void imbue(const locale& loc) ;              // задать локализацию
  virtual basic_streambuf* setbuf(Ch* p, streamsize n) ;

  virtual pos_type seekoff(off_type off, ios_base::seekdir way,
                  ios_base::openmode m = ios_base::in|ios_base::out) ;
  virtual pos_type seekpos(pos_type p,
                  ios_base::openmode m = ios_base::in|ios_base::out) ;

  virtual int sync() ;                                  // ввод sync() (§21.6.2)

  virtual int showmanyc() ;
  virtual streamsize xsgetn(Ch*p, streamsize n) ;       // получить n символов

  virtual int_type underflow() ;
  virtual int_type uflow() ;

  virtual int_type pbackfail(int_type c = Tr::eof()) ;

  virtual streamsize xsputn(const Ch* p, streamsize n) ; // записать n символов
  virtual int_type overflow(int_type c = Tr::eof()) ;    // область записи заполнена
};
```

Функции ***underflow***() и ***uflow***() вызываются для того, чтобы получить символ из реального источника, когда буфер пуст. Если у источника нет символа для ввода, поток устанавливается в состояние ***eof*** (§21.3.3), и если при этом исключение не генерируется, то возвращается ***traits_type::eof***(). После возврата символа функцией ***uflow***() (но не функцией ***underflow***()) ***gptr***() инкрементируется. Помните, что в типичном случае в вашей системе буферов больше, чем их имеется в потоках ***iostream***, так что могут иметь место задержки даже в случае использования небуферизованных потоков ввода/вывода.

Функция ***overflow***() вызывается для передачи символов на реальное устройство по заполнении буфера. Вызов ***overflow***(***c***) выводит содержимое буфера плюс символ ***c***. Если вывод на устройство невозможен, поток устанавливается в состояние ***eof*** (§21.3.3), и если при этом исключение не генерируется, то возвращается ***traits_type***::***eof***().

Функция ***showmanyc***() — «show how many characters» (показать, как много символов) — довольно странная функция, призванная помочь пользователю справиться о состоянии машинной системы ввода. Она возвращает оценочное число символов, которые могут быть введены путем, например, сброса буферов операционной системы, а не ожиданием ввода с диска. Вызов ***showmanyc***() возвращает ***-1*** в случае, когда нельзя обещать ввода даже одного символа до возможной встречи с концом файла. Это очень низкоуровневая функция, сильно зависящая от реализации. Не пытайтесь ее использовать, не ознакомившись детально с документацией по вашей системе и не проведя некоторое количество экспериментов.

По умолчанию, потоки используют глобально установленные характеристики локализации (интернационализации) (§21.7). Вызовы ***pubimbue***(***loc***) или ***imbue***(***loc***) установят локализацию потоков с помощью объекта ***loc***.

Для конкретного вида потока ***streambuf*** наследуется от ***basic_streambuf***. Он предоставляет конструкторы и функции инициализации, привязывающие ***streambuf*** к конкретному источнику символов или устройству их ввода, и замещает виртуальные функции, определяющие стратегию буферизации. Например:

```
template<class Ch, class Tr = char_traits<Ch> >
class basic_filebuf: public basic_streambuf<Ch, Tr>
{
public:
  basic_filebuf() ;
  virtual ~basic_filebuf() ;

  boot is_open() const;
  basic_filebuf* open(const char* p, ios base::openmode mode) ;
  basic_filebuf* close() ;

protected:
  virtual int showmanyc() ;
  virtual int_type underflow() ;
  virtual int_type uflow() ;
  virtual int_type pbackfail(int_type c = Tr::eof())
  virtual int_type overflow(int_type c = Tr::eof()) ;

  virtual basic_streambuf<Ch, Tr>* setbuf(Ch* p, streamsize n) ;
  virtual pos_type seekoff(off_type off, iosbase::seekdir way,
                    ios_base::openmode m = ios_base::in|ios_base::out) ;
  virtual pos_type seekpos(pos_type p,
                    ios_base::openmode m = ios_base::in|ios_base::out)
  virtual int sync() ;
  virtual void imbue(const locale& loc) ;
};
```

Функции для манипуляции буферами и др. наследуются без каких-либо изменений от ***basic_streambuf***. Только функции, влияющие на инициализацию и стратегию буферизации должны определяться отдельно.

Как обычно, с помощью операторов ***typedef*** предоставляются наиболее распространенные и очевидные типы:

```
rypedef basic_streambuf<char> streambuf;
typedef basic_stringbuf<char> stringbuf;
typedef basic_filebuf<char>   filebuf;

typedef basic_streambuf<wchar_t> wstreambuf;
typedef basic_stringbuf<wchar_t> wstringbuf;
typedef basic_filebuf<wchar_t>   wfilebuf;
```

21.7. Локализация (интернационализация)

Класс ***locale*** отвечает за локализацию национально зависимых характеристик, таких как классификация символов на буквы, цифры и т.п.; он также имеет отношение к правилам сортировки строк и к внешнему виду представления числовых значений для ввода и вывода. Чаще всего, класс ***locale*** используется в потоковой библиотеке ввода/вывода неявным образом, обеспечивая стандартные соглашения

для некоторой группы естественных языков (в первую очередь — английского языка). Однако изменяя ***locale*** для потока, программист может «взять на себя» управление вопросами интернационализации/локализации программного обеспечения, обеспечив выполнение специфических языковых правил.

Класс ***locale*** определен в пространстве имен ***std*** и представлен в заголовочном файле <***locale***> (§D.2):

```
class locale                             // полное объявление см. в §D.2
{
  // ...
  locale () throw () ;                   // копия текущей глобальной локализации
  explicit locale (const char* name) ;   // создать локализацию по имени С-локализации
  basic_string<char> name () const;      // получить имя данной локализации

  locale (const locale&) throw () ;      // копировать локализацию
  const locale& operator= (const locale&) throw () ; // копировать локализацию

  static locale global (const locale&) ; // установить глобальную локализацию
  static const locale& classic () ;      // получить локализацию языка С
};
```

В простейшем случае класс ***locale*** используется для того, чтобы переключить языковые правила с текущих на иные. Например:

```
void f()
{
  std::locale loc ("POSIX") ; // стандартная локализация для POSIX
  cin.imbue(loc) ;             // пусть cin использует loc
  // ...
  cin.imbue(std::locale()) ; // возвращаем для cin умолчательную глобаль. локализацию
}
```

Функция ***imbue***() является членом класса ***basic_ios*** (§21.7.1). Как показано в примере, некоторые наиболее распространенные правила интернационализации имеют зарезервированные строковые имена. Они, как правило, коррелируют в предпочтениях с языком С.

Можно зафиксировать параметры локализации так, чтобы они по умолчанию использовались всеми создаваемыми потоками:

```
void g (const locale& loc = locale ())        // текущая умолчательная глобаль. локал-я
{
  locale old_global = locale::global(loc) ; // теперь loc будет умолчательной локал-ией
  // ...
}
```

Установка глобальных языковых характеристик не влияет на существующие потоки. В частности, это не оказывает влияния на ***cin***, ***cout*** и т.д. Для влияния на эти потоки к ним нужно применить функцию ***imbue***(). Это изменяет многие детали их поведения.

Можно использовать члены класса ***locale*** напрямую, можно определять новые объекты локализации, можно дополнять их новыми особенностями. Например, их можно применить для изменения обозначения денежных единиц при вводе/выводе, формата дат и т.п. (§21.10[25]), а также для преобразования кодировок. Концеп-

ция объектов локализации, классы ***locale*** и ***facet***, а также стандартные объекты этих классов рассматриваются в приложении D.

Локализация в стиле языка С представлена в заголовочных файлах <***clocale***> и <***locale.h***>.

21.7.1. Функции обратного вызова для потоков

Иногда требуется что-то добавить к состоянию потока. Например, кому-то может потребоваться, чтобы поток «знал», в каком виде выводить числа типа ***complex*** — в полярных или декартовых координатах. Класс ***ios_base*** предоставляет функцию ***xalloc***() для выделения памяти под простую информацию о состоянии потока. Возврат функции ***xalloc***() идентифицирует местоположение, к которому можно обратиться с помощью функций ***iword***() и ***pword***():

```
class ios_base
{
public:
  // ...
  ~ios_base();

  locale imbue(const locale& loc);   // установить локализ-ю (вернуть старую) (§D.2.3)
  locale getloc() const;             // получить локализацию

  static int xalloc();               // получить целое и указатель (инициализированы 0)

  long& iword(int i);                // доступ к целому iword(i)
  void*& pword(int i);               // доступ к указателю pword(i)

  // обратные вызовы:
  enum event {erase_event, imbue_event, copyfmt_event};  // типы событий

  typedef void (*event_callback)(event, ios_base&, int i);
  void register_callback(event_callback f, int i);        // прикрепить f к pword(i)
};
```

Иногда разработчику реализации или пользователю нужно знать об изменении состояния потока. Функция ***register_callback***() регистрирует функцию, которая будет вызвана при наступлении «ее события». В итоге, при вызовах ***imbue***(), ***copyfmt***() или ~***ios_base***() будут вызваны функции, зарегистрированные для событий ***imbue_event***, ***copyfmt_event*** или ***erase_event***, соответственно. При изменении состояния зарегистрированные функции вызываются с аргументом ***pword***(***i***), где ***i*** соответствует их ***register_callback***().

Эти механизмы хранения и обратного вызова довольно запутанные. Применяйте их лишь при крайней необходимости в расширении возможностей форматирования нижнего уровня.

21.8. Ввод/вывод языка С

Поскольку код С++ и код С часто перемешиваются в рамках одного проекта, потоки ввода/вывода С++ сосуществуют с семейством функций ***printf***() языка С. Функции ввода/вывода в стиле С представлены в заголовочных файлах <***cstdio***> и <***stdio.h***>. Поскольку С-функции можно вызывать из С++-кода, некоторые про-

граммисты любят использовать более привычные им функции ввода/вывода языка С. Даже если вы сами предпочитаете работать с потоковым вводом/выводом, вы все равно когда-нибудь столкнетесь со вводом/выводом в стиле языка С.

Ввод/вывод языков С и С++ может быть смешанным — вызов *sync_with_stdio*() до применения в программе операций потокового ввода/вывода гарантирует, что операции ввода/вывода языков С и С++ будут использовать общие буферы. В то же время, вызов *sync_with_stdio*(*false*) в том же контексте предотвращает совместное использование буферов и может улучшить производительность ввода/вывода:

```
class ios_base
{
  // ...
  static bool sync_with_stdio (bool sync = true) ;
};
```

Главным преимуществом потоковых функций вывода перед функцией *printf*() из стандартной библиотеки языка С состоит в том, что они безопасны по отношению к разным типам, и что вывод с их помощью встроенных и пользовательских типов выполняется в едином стиле.

Основные функции языка С для вывода

```
int printf(const char* format...) ;               // пишет в stdout
int fprintf(FILE*, const char* format...) ;       // пишет в "file" (stdout, stderr)
int sprintf(char* p, const char* format...) ;     // пишет в p[0] ...
```

выполняют форматированный вывод произвольной последовательности аргументов под управлением форматирующей строки *format*. Форматирующая строка содержит два вида элементов — обычные подлежащие выводу символы и форматирующие спецификаторы, каждый из которых вызывает выполнение преобразования и вывод очередного аргумента. Форматирующие спецификаторы начинаются символом %. Например:

```
printf("there were %d members present.", no_of_members) ;
```

Здесь *%d* означает, что аргумент *no_of_members* должен трактоваться как *int* и выводиться в виде соответствующей последовательности десятичных цифр. Если, например, *no_of_members==127*, то на выходе получим:

```
there were 127 members present.
```

Набор спецификаций преобразований довольно велик и обеспечивает высокую степень гибкости. После знака % может присутствовать следующее:

- необязательный знак минус, который специфицирует выравнивание влево преобразованного значения внутри поля;

\+ необязательный знак плюс, который требует, чтобы значения знаковых типов выводились с лидирующими + или –;

0 необязательный *нуль*, который приводит к тому, что выводимое числовое значение дополняется лидирующими нулями. Если установлена точность вывода или указана спецификация -, то этот *нуль* игнорируется;

необязательный знак (решетка), означающий: что числа с плавающей запятой выводятся с точкой, даже если после точки значащих цифр нет; что выводятся «хвостовые» нули; что восьмеричные значения выводятся с лидирующим нулем; что шестнадцатеричные значения предваряются ***0x*** или ***0X***;

d необязательная последовательность цифр, задающая ширину поля; если преобразованное число имеет меньшее число символов, чем ширина поля, то поле будет дополнено пробелами слева (или справа — если указан индикатор выравнивания влево); если число ***d*** указано с начальным нулем, то заполнение вместо пробелов выполняется нулями;

. необязательная точка, которая служит для отделения ширины поля от последующей строки цифр;

d необязательная последовательность (строка) цифр, задающая точность, и которая означает число цифр после десятичной точки для ***e*** и ***f*** форматов, или максимальное число выводимых символов;

* ширина поля или точность могут быть указаны звездочкой — в этом случае их величину задает целочисленный аргумент;

h необязательный символ, указывающий, что последующие ***d***, ***o***, ***x*** или ***u*** относятся к целочисленному аргументу типа ***short***;

l необязательный символ, указывающий, что последующие ***d***, ***o***, ***x*** или ***u*** относятся к целочисленному аргументу типа ***long***;

L необязательный символ, указывающий, что последующие ***e***, ***E***, ***g***, ***G*** или ***f*** относятся к типу ***double***;

% этот символ требует вывести символ % (никакие аргументы при этом не используются);

c символ, указывающий тип преобразования; применяются следующим символы преобразований:

- d Целый аргумент преобразуется к десятичному виду;
- i Целый аргумент преобразуется к десятичному виду;
- o Целый аргумент преобразуется к восьмеричному виду;
- x Целый аргумент преобразуется к шестнадцатеричному виду;
- X Целый аргумент преобразуется к шестнадцатеричному виду;
- f Аргумент типа ***float*** или ***double*** преобразуется к десятичному виду [-]*ddd.ddd*; количество цифр после запятой соответствует точности, указанной для этого аргумента; при необходимости число округляется; если точность не задана, выводится 6 цифр; если для точности указан ***0***, а # не используется, то ни цифры, ни точка не выводятся;
- e Аргумент типа ***float*** или ***double*** преобразуется к десятичной нотации в научном стиле [-]*d.ddde+dd* или [-]*d.ddde-dd*, где до точки стоит одна цифра, а количество цифр после точки соответствует указанной для аргумента точности; при необходимости число округляется; если точность не задана, выводится 6 цифр; если для точности указан ***0***, а # не используется, то ни цифры, ни точка не выводятся;

Совпадает с предыдущим случаем, но буква *e* выводится в верхнем регистре — как *E*;

Аргумент типа ***float*** или ***double*** выводится в стиле ***d***, ***f*** или ***e*** в зависимости от того, что дает максимальную точность при минимуме занимаемого места;

Совпадает с предыдущим случаем, но буква *e* выводится в верхнем регистре — как *E*;

c Выводится символьный аргумент; нулевые символы игнорируются;

s Аргумент трактуется как *строка* (указатель на символы), и символы строки выводятся до тех пор, пока не встретится нулевой символ, или пока не исчерпается количество символов, указанное точностью; если точность отсутствует или указана как *0*, то вывод продолжается до нулевого символа;

p Аргумент трактуется как указатель; конкретный вид представления значения указателя зависит от реализации;

u Целый аргумент без знака преобразуется к десятичной нотации;

n Количество символов, выведенное к этому моменту функциями ***printf***(), ***fprintf***() или ***sprintf***(), записывается в аргумент типа ***int***, указуемый через указатель типа ***int****.

Не указанная или недостаточная ширина поля не приводят к урезанию значений; дополнение символами-заполнителями имеет место тогда, когда указанная ширина поля превосходит необходимую ширину.

Вот более сложный пример

```
char* line_format = "#line %d \"%s\"\n";
int line = 13;
char* file_name = "C++/main.c";

printf("int a;\n");
printf(line_format, line, file_name);
```

в результате чего будет выведено

```
int a;
#line 13 "C++/main.c"
```

Использование ***printf***() небезопасно в том смысле, что никакой проверки типа при этом не выполняется. Вот известный пример, как получить непредсказуемый вывод, вывод некоторого участка памяти или чего похуже:

```
char x = 'q';
printf("bad input char: %s" ,x); // вместо %s должно быть %c
```

В то же время, функция ***printf***() обеспечивает чрезвычайную гибкость, и она хорошо знакома программирующим на языке С.

Аналогично, ***getchar***() часто применяется для ввода символов:

```
int i;
while((i=getchar()) != EOF) {/* испол уем i */}
```

Чтобы можно было проверять конец ввода посредством сравнения с целочисленным ***EOF***, возврат ***getchar***() нужно сохранять в переменной типа ***int***, а не ***char***.

Подробнее со вводом/выводом в стиле языка С можно ознакомиться по справочной литературе или по книге «The C Programming Language» [Kernighan, Ritchie, 1988].

21.9. Советы

1. Определяйте операции << и >> для пользовательских типов с очевидными текстовыми представлениями их значений; §21.2.3, §21.3.5.
2. При выводе выражений с операциями, имеющими низкий приоритет, пользуйтесь круглыми скобками; §21.2.
3. Для добавления операций << или >> нет необходимости в модификации ***istream*** или ***ostream***; §21.2.3.
4. Вы можете определить функцию, которая ведет себя как виртуальная по второму аргументу; §21.2.3.1.
5. Помните, что по умолчанию операция >> пропускает пробельные символы; §21.3.2.
6. Используйте низкоуровневые функции ввода вроде ***get***() и ***read***() в основном при реализации высокоуровневых функций ввода; §21.3.4.
7. Используя ***get***(), ***getline***() и ***read***(), будьте внимательны в отношении критерия конца ввода; §21.3.4.
8. Для управления вводом/выводом вместо флагов состояния применяйте манипуляторы; §21.3.3, §21.4, §21.4.6.
9. Применяйте исключения только для перехвата редких ошибок ввода/вывода; §21.3.6.
10. Связывайте потоки, используемые для интерактивного ввода и вывода; §21.3.7.
11. Используйте часовых для концентрации начального и завершающего кода многих функций в одном месте; §21.3.8.
12. Не применяйте круглые скобки для манипуляторов без аргументов; §21.4.6.2.
13. Используя стандартные манипуляторы, не забывайте включать #***include*** <***iomanip***>; §21.4.6.2.
14. Вы можете достичь эффекта (и эффективности) тернарной операции, определив простой функциональный класс; §21.4.6.3.
15. Помните, что спецификация ***width*** относится лишь к ближайшей операции ввода/вывода; §21.4.4.
16. Помните, что спецификация ***precision*** относится ко всем последующим операциям вывода чисел с плавающей запятой; §21.4.3.
17. Используйте строковые потоки для форматирования в памяти; §21.5.3.
18. Вы можете задать режим работы файлового потока; §21.5.1.
19. Расширяя систему ввода/вывода, четко различайте форматирование (***iostream***) и буферизацию (***streambuf***); §21.1, §21.6.

20. Реализуйте нестандартные способы передачи значений через буферы потоков; §21.6.4.
21. Реализуйте нестандартные способы форматирования значений через операции над потоками; §21.2.3, §21.3.5.
22. Вы можете изолировать и инкапсулировать обращение к пользовательскому коду с помощью соответствующей пары функций; §21.6.4.
23. Вы можете использовать ***in_avail***() для того, чтобы заранее убедиться в возможности операции ввода; §21.6.4.
24. Различайте простые эффективные операции и операции, реализующие стратегию: первые делайте встраиваемыми, а вторые — виртуальными; §21.6.4.
25. Для отражения национальных особенностей применяйте класс ***locale***; §21.7.
26. Для смешения или разъединения буферов ввода/вывода языков С и С++ вызывайте ***sync_with_stdio***(); §21.8.
27. Применяя ввод/вывод в стиле С, помните об ошибках, связанных с типами; §21.8.

21.10. Упражнения

1. (*1.5) Прочитайте файл, содержащий числа с плавающей запятой, скомбинируйте пары прочитанных чисел в комплексные значения и выведите их.
2. (*1.5) Определите тип ***Name_and_address***. Определите операции << и >> для этого типа. Скопируйте поток объектов типа ***Name_and_address***.
3. (*2.5) Скопируйте поток объектов типа ***Name_and_address*** и вставьте в него столько ошибок, сколько сумеете придумать (например, ошибки форматирования, или ошибочное определение конца строк). Обработайте эти ошибки таким образом, чтобы функция копирования смогла прочитать большинство из корректно отформатированных объектов типа ***Name_and_address***, несмотря на то, что на входе «хорошие» значения будут чередоваться с «плохими».
4. (*2.5) Переопределите формат ввода/вывода объектов типа ***Name_and_address*** так, чтобы он стал менее чувствительным к ошибкам ввода.
5. (*2.5) Разработайте ряд функций для запроса и чтения информации различного типа. Подсказка: целые числа, числа с плавающей запятой, имена файлов, почтовые адреса, даты, персональная информация и т.д. Попытайтесь защитить функции от ошибок ввода («защита от дурака»).
6. (*1.5) Напишите программу, которая выводит: все буквы нижнего регистра, все буквы, все буквы и цифры, все символы из идентификаторов С++ вашей системы, все знаки препинания, числовые коды управляющих символов, все терминальные символы, все коды терминальных символов и, наконец, все печатные символы.
7. (*2) Читайте строки текста в символьный буфер фиксированного размера. Удалите все пробельные символы и замените все алфавитные символы на следующие за ними в алфавите (***z*** на ***a***, ***9*** на ***0***). Выведите получившуюся строку.

8. (*3) Напишите миниатюрную систему потокового ввода/вывода, предоставляющую классы ***istream***, ***ostream***, ***ifstream***, ***ofstream***, функции ***operator***<<() и ***operator***>>() для целых чисел и такие операции, как ***open***() и ***close***(), для файлов.
9. (*4) Реализуйте стандартную библиотеку ввода/вывода языка С (<***stdio.h***>) при помощи стандартной библиотеки ввода/вывода языка С++ (<***iostream***>).
10. (*4) Реализуйте стандартную библиотеку ввода/вывода языка С++ (<***iostream***>) при помощи стандартной библиотеки ввода/вывода языка С (<***stdio.h***>).
11. (*4) Реализуйте эти библиотеки так, чтобы их можно было использовать одновременно.
12. (*2) Реализуйте класс с перегруженной операцией [], предназначенной для чтения файлов по заданной в нем позиции.
13. (*3) Повторите предыдущее упражнение, но сделайте операцию [] пригодной как для чтения, так и для записи. Намек: сделайте так, чтобы операция [] возвращала дескриптор, для которого присваивание означало бы запись в файл, а неявное преобразование к ***char*** — чтение из файла.
14. (*2) Повторите предыдущее упражнение, но сделайте операцию [] пригодной для объектов разных типов, а не только для символов типа ***char***.
15. (*3.5) Реализуйте версии ***istream*** и ***ostream***, которые читают и записывают числа в бинарном формате, а не переводят их в символы. Обсудите достоинства и недостатки такого подхода по сравнению со стандартным (символьным) подходом.
16. (*3.5) Разработайте и реализуйте операцию ввода по шаблону. Используйте форматирующие строки в стиле функции ***printf***() для задания шаблона ввода. Должна иметься возможность применения к одному и тому же вводу разных шаблонов для установления истинного формата. Можно реализовать класс как производный от ***istream***.
17. (*4) Придумайте и реализуйте наилучший вид шаблона для шаблонного ввода. Уточните, в чем именно состоит его преимущество.
18. (*2) Определите манипулятор вывода ***based***, имеющий два аргумента — систему счисления и целое значение, и выводящий целое число в соответствии с указанной системой счисления. Например, ***based***(***2***, ***9***) должно вывести ***1001***.
19. (*2) Напишите манипуляторы, которые включают и выключают эхо-повторы символов при их вводе.
20. (*2) Реализуйте ***Bound_form*** из §21.4.6.3 для обычного набора встроенных типов.
21. (*2) Переопределите ***Bound_form*** из §21.4.6.3 так, чтобы операция вывода никогда не выходила за установленную ширину (***width***(). Нужно также гарантировать, что выводимое значение не будет незаметным образом усечено по точности.

22. (*3) Реализуйте манипулятор ***encrypt*(*k*)**, который шифрует вывод в поток ***ostream*** с ключом ***k***. Реализуйте также противоположный по смыслу манипулятор ***decrypt*()** для чтения из потока ***istream***. Предоставьте средство отключения шифрования, чтобы последующий вывод шел открытым текстом.
23. (*2) Проследите путь символа в вашей системе от клавиатуры до экрана на примере следующего кода:

```
char c;
cin >> c;
cout<< c << end;
```

24. (*2) Модифицируйте ***readints*()** (§21.3.6) так, чтобы обрабатывались все исключения. Подсказка: «выделение ресурса есть инициализация».
25. (*2.5) Существуют стандартные способы чтения, записи и представления дат под управлением класса ***locale***. Прочитайте о них в документации к вашей реализации и напишите небольшую программу чтения и записи дат при помощи этого механизма. Подсказка: ***struct tm***.
26. (*2.5) Определите поток вывода с именем ***ostrstream***, который можно было бы прикреплять к массиву символов (С-строке) таким же способом, каким ***ostringstream*** прикрепляется к строкам типа ***string***. Однако не копируйте массив в или из ***ostrstream***. Поток ***ostrstream*** должен просто обеспечить возможность записи в прикрепленный к нему массив символов. Его можно было бы использовать для форматирования в памяти следующим образом:

```
char buf[message_size];
ostrstream ost(buf, message_size);
do_something(arguments, ost);      // вывод в буфер buf через ost
cout << buf;                       // ost adds добавляет терминальный 0
```

Операция вроде ***do_something*()** может писать в поток ***ost***, передавать поток ***ost*** другим операциям и т.д., используя стандартные операции вывода. Нет необходимости контролировать переполнение, ибо ***ost*** знает свои размеры и переходит в состояние ***fail*()** при его заполнении. Операция ***display*()** может писать сообщения в «настоящий» поток вывода. Такой прием особо удобен, когда конечная операция должна осуществлять вывод на устройство, отличное от традиционных строкоориентированных устройств вывода. Например, может понадобиться помещать текст из ***ost*** в область экрана с фиксированными размерами. В конце аналогичным образом определите и потоковый класс ***istrstream***, который читает строки из символьного массива с терминальным нулем. Интерпретируйте терминальный нуль как символ end-of-file («конец файла»). Такие строковые потоки входили в ранние варианты библиотеки потоков и их обычно можно найти в ***<strstream.h>***.
27. (*2.5) Реализуйте манипулятор ***general*()**, который возвращает поток в его оригинальное состояние форматирования так, как ***scientific*()** (§21.4.6.2) заставляет поток использовать научный формат.

22

Классы для математических вычислений

Цель вычислений — понимание, а не числа.
— Р.В. Хэмминг

...но для студента числа — наилучший путь к пониманию.
— А. Ролстон

Введение — предельные значения — стандартные математические функции — класс ***valarray*** — векторные операции — срезы — массив ***slice_array*** — устранение временных объектов — ***gslice_array*** — ***mask_array*** — ***indirect_array*** — комплексная арифметика — обобщенные численные алгоритмы — случайные числа — советы — упражнения.

22.1. Введение

Трудно написать какой-либо реальный код, не содержащий изрядную долю вычислений. Но чаще всего серьезных математических вычислений не требуется. Настоящая глава рассматривает те средства стандартной библиотеки, которые идут много дальше простейшей арифметики.

Ни язык С, ни С++ не разрабатывались намеренно с целью поддержки выполнения разветвленных математических расчетов. Однако серьезные математические расчеты все равно так или иначе возникают в контексте самых разных прикладных задач — доступ к базам данных, работа с сетями, управление внешними устройствами, графика, моделирование, финансовый анализ и т.д. В итоге, на С++ приходится выполнять немалый объем математических вычислений, составляющих определенную часть более крупных проектов. В настоящее время характерные математические вычисления выходят далеко за рамки простых циклических расчетов с наборами чисел с плавающей запятой. Но там, где встречаются более сложные структуры данных, мощь языка С++ становится очевидной, и поэтому научные и инженерные расчеты

все чаще стали выполняться с применением языка C++. В настоящей главе рассматриваются те части стандартной библиотеки C++, которые предназначены для поддержки математических вычислений. Рассматривается ряд приемов, характерных для программ, в которых математические расчеты формулируются на языке C++. Однако я не пытаюсь учить здесь математике или численным методам, которые сами по себе являются захватывающей темой. Чтобы вникнуть в эту тему, вам потребуется хороший курс по численным методам или, по крайней мере, учебник по этому предмету, а не руководство по языку программирования.

22.2. Числовые пределы

Чтобы уверенно работать с числами, нужно хорошо знать общие свойства встроенных числовых типов, а они зависят больше от реализации, чем от правил языка программирования (§4.6). Например, каково наибольшее значение для типа ***int***? Каково наименьшее значение для ***float***? Преобразование от ***double*** к ***float*** вызывает усечение или округление? Сколько бит в ***char***?

Ответы на подобного рода вопросы дают специализации шаблона ***numeric_limits***, представленного в заголовочном файле <***limits***>. Например:

```
void f(double d, int i)
{
  if(numeric_limits<unsigned char>::digits != 8)
  {
    // необычные байты (число бит не равно 8)
  }
  if(i<numeric_limits<short>::min() || numeric_limits<short>::max()<i)
  {
    // i не может храниться в short без потери точности
  }
  if(0<d && d<numeric_limits<double>::epsilon()) d=0;
  if(numeric_limits<Quad>::is_specialized)
  {
    // доступна информация о числовых пределах для типа Quad
  }
}
```

Каждая специализация предоставляет стержневую информацию по типу ее аргумента, так что на долю общего шаблона ***numeric_limits*** остаются определения констант и простых встраиваемых функций:

```
template<class T> class numeric limits
{
public:
  static const bool is_specialized = false; // есть информация для numeric_limits<T>?
  // неинтересные умолчательные установки
};
```

Реальная информация сосредоточена в специализациях шаблона. Каждая реализация обязана предоставить по специализации этого шаблона для каждого фунда-

ментального типа (символьных типов, типа ***bool***, целых чисел и чисел с плавающей запятой), но не для таких типов, как ***void***, перечисления или библиотечные типы (например, ***complex***<***double***>).

Для интегральных типов, таких как ***char***, интерес представляет небольшая часть информации. Вот специализация ***numeric_limits***<***char***> для реализации, в которой тип ***char*** знаковый и содержит 8 бит:

```
template<> class numeric_limits<char>
{
public:
  static const bool is_specialized = true;     // да, информация есть

  static const int digits = 7;                 // число бит (исключая знак)

  static const bool is_signed = true;          // в данной реализации char знаковый
  static const bool is_integer = true;         // char - это интегральный тип

  static char min() throw() {return -128; }    // наименьшее значение
  static char max() throw() {return 127; }     // наибольшее значение

  // масса объявлений, не имеющих отношение к char
};
```

Заметьте, что для целого знакового типа ***digits*** на единицу меньше, чем число бит в этом типе.

Большинство членов ***numeric_limits*** предназначены для описания чисел с плавающей запятой. Например, вот одна из возможных реализаций для ***float***:

```
template<> class numeric_limits<float>
{
public:
    static const bool is_specialized = true;

    static const int radix = 2;      // основание степени (в данном случае, двойки)
    static const int digits = 24;    // количество (двоичных) цифр в мантиссе
    static const int digits10 = 6;   // количество десятичных цифр в мантиссе

    static const bool is_signed = true;
    static const bool is_integer = false;
    static const bool is_exact = false;

    static float min() throw() {return 1.17549435E-38F; }
    static float max() throw() {return 3.40282347E+38F; }

    static float epsilon() throw() {return 1.19209290E-07F; }
    static float round_error() throw() {return 0.5F; }

    static float infinity() throw() {return /* некоторое значение */; }
    static float quiet_NaN() throw() {return /* некоторое значение */; }
    static float signaling_NaN() throw() {return /* некоторое значение */; }
    static float denorm_min() throw() {return min(); }

    static const int min_exponent = -125;
    static const int min_exponent10 = -37;
    static const int max_exponent = +128;
    static const int max_exponent10 = +38;
```

```
    static const bool has_infinity = true;
    static const bool has_quiet_NaN = true;
    static const bool has_signaling_NaN = true;
    static const float_denorm_style has_denorm = denorm_absent;  // enum из <limits>
    static const bool has_denorm_loss = false;

    static const bool is_iec559 = true;    // отвечает IEC-559
    static const bool is_bounded = true;
    static const bool is_modulo = false;
    static const bool traps = true;
    static const bool tinyness_before = true;

    static const float_round_style round_style = round_to_nearest;  // enum из <limits>
};
```

Отметим, что *min*() — это самое маленькое положительное нормализованное число, а *epsilon* — самое маленькое положительное число с плавающей запятой такое, что *1+epsilon-1* больше нуля.

Определяя некоторый скалярный тип в духе встроенных типов, неплохо определить для него и подходящую специализацию *numeric_limits*. Например, если бы я написал тип *Quad* для чисел с учетверенной точностью, или если бы в реализации оказался тип *long long*, пользователь мог бы ожидать наличия специализации *numeric_limits*<*Quad*> или *numeric_limits*<*long long*>.

Теоретически можно представить себе специализацию *numeric_limits* для пользовательского типа, имеющего мало общего с числами с плавающей запятой. Но в таких случаях лучше использовать обычные способы для представления свойств типа, а не городить специализацию *numeric_limits* со свойствами, отличными от стандартных.

В *numeric_limits* свойства чисел с плавающей запятой представляются с помощью встраиваемых функций. Для интегральных же типов свойства должны представляться в форме, допускающей их использование в константных выражениях. Это значит, что они должны иметь инициализаторы внутри класса (§10.4.6.2). Если вы для этого используете статические константные члены, а не перечисления, не забудьте определить эти статические члены.

22.2.1. Макросы для предельных значений

От языка C язык C++ унаследовал макросы, описывающие свойства целых чисел. Они расположены в заголовочных файлах <*climits*> и <*limits.h*> и имеют имена вроде *CHAR_BIT* и *INT_MAX*. Аналогично для чисел с плавающей запятой в заголовочных файлах <*cfloat*> и <*float.h*> расположены макросы вроде *DBL_MIN_ EXP*, *FLT_RADIX* и *LDBL_MAX*.

Как всегда, применения макросов лучше избегать.

22.3. Стандартные математические функции

Заголовочные файлы <*cmath*> и <*math.h*> объявляют то, что принято называть «общепринятыми математическими функциями»:

```
double abs (double) ;                    // абсолютное значение (не в C); то же, что fabs()
double fabs (double) ;                   // абсолютное значение

double ceil (double d) ;                 // наименьшее целое, не меньшее d
double floor (double d) ;                // наибольшее целое, не большее d

double sqrt (double d) ;                 // квадратный корень из d, (d не отрицательное)

double pow (double d, double x) ;        // d в степени x, (error если d==0 и x<=0 и т.п.)
double pow (double d, int i) ;           // d в степени i (не в C)

double cos (double) ;                    // косинус
double sin (double) ;                    // синус
double tan (double) ;                    // тангенс

double acos (double) ;                   // арккосинус
double asin (double) ;                   // арксинус
double atan (double) ;                   // арктангенс
double atan2 (double x, double y) ;      // atan(x/y)

double sinh (double) ;                   // гиперболический синус
double cosh (double) ;                   // гиперболический косинус
double tanh (double) ;                   // гиперболический тангенс

double exp (double) ;                    // экспонента, основание e
double log (double d) ;                  // натуральный логарифм (d должно быть >0)
double log10 (double d) ;                // десятичный логарифм

double modf (double d, double* p) ;      // возврат дробной части d (целая часть в *p)
double frexp (double d, int* p) ;        // ищет x в [.5,1) а y так что d = x*pow(2,y),
                                         // возвращает x и сохраняет y в *p
double fmod (double d, double m) ;       // остаток от деления (знак как у d)
double ldexp (double d, int i) ;         // d*pow(2,i)
```

Кроме того, заголовочные файлы <***cmath***> и <***math.h***> предоставляют варианты этих функций для аргументов типа ***float*** и ***long double***.

Для функций, результат которых в общем случае неоднозначен — например ***asin***() — возвращается ближайшее к нулю значение. Функция ***acos***() возвращает неотрицательное значение.

Когда возникают ошибки, ***errno*** из <***cerrno***> принимает значение ***EDOM*** в случае выхода аргумента из области определения функций, и значение ***ERANGE*** — в случае выхода результата за возможные пределы значений типа. Например:

```
void f()
{
  errno = 0;              // очистка старого состояния, инициирующего ошибку
sqrt(-1) ;
  if(errno==EDOM) cerr << "sqrt() not defined for negative argument";

  pow(numeric_limits<double>::max(),2) ;
  if(errno == ERANGE) cerr << "result of pow() too large to represent as a double";
}
```

По историческим причинам некоторые математические функции объявляются в <***cstdlib***>, а не в <***cmath***>:

```
int abs (int) ;                          // абсолютное значение
long abs (long) ;                        // абсолютное значение (не в С)
long labs (long) ;                       // абсолютное значение

struct div_t  {implementation_defined quot, rem; } ;
struct ldiv_t {implementation_defined quot, rem; } ;

div_t div (int n, int d) ;               // деление n на d, возвращает (частное,остаток)
ldiv_t div (long int n, long int d) ;    // деление n на d, (не в С)
ldiv_t ldiv (long int n, long int d) ;   // деление n на d, возвращает (частное,остаток)
```

22.4. Векторная арифметика

Часто в программах математические вычисления сводятся к относительно простой обработке одномерных массивов (векторов) чисел с плавающей запятой. Такие векторы, в частности, поддерживаются на аппаратном уровне для высокопроизводительных суперкомпьютеров, для них написано множество библиотек общего назначения, и, кроме того, имеются их глубоко оптимизированные версии для специальных применений. Как следствие, стандартная библиотека языка C++ также предоставляет специальный *векторный класс* — ***valarray*** — *специально разработанный для быстрых численных операций над векторами.*

Глядя на предоставляемые классом ***valarray*** средства, нужно помнить, что они представляют собой довольно низкоуровневые строительные блоки для высокопроизводительных вычислений. В итоге, главным критерием проектирования была не простота их использования, а возможность эффективного применения на высокопроизводительных компьютерах в условиях агрессивной оптимизации. Если ваша цель — это гибкость и универсальность, а вовсе не особая эффективность, то вам лучше строить вычисления на основе контейнеров из глав 16 и 17, и не пытаться втиснуться в жесткие рамки простого, эффективного и намеренно традиционного вектора ***valarray***.

Кто-нибудь может сказать, что ***valarray*** следовало бы назвать ***vector***, поскольку он близок традиционному математическому вектору, а ***vector*** (§16.3) следовало бы назвать ***array***, так как это скорее массив в обычном смысле. Но терминология развивалась в ином направлении: ***valarray*** — это специальный вектор, оптимизированный для серьезных математических вычислений; ***vector*** — это гибкий контейнер для хранения и манипулирования объектами различных типов, а массивами (arrays) принято называть низкоуровневые встроенные типы данных.

Тип ***valarray*** поддержан четырьмя подтипами, специфицирующими подмножества valarray:

- ***slice_array*** и ***gslice_array*** представляют понятие срезов (§22.4.6, §22.4.8),
- ***mask_array*** специфицирует подмножество путем маркировки элементов (§22.4.9),
- ***indirect_array*** содержит индексы необходимых элементов (§22.4.10).

22.4.1. Конструкторы класса valarray

Сам класс ***valarray*** и его вспомогательные средства определены в пространстве имен ***std*** и представлены в заголовочном файле <***valarray***>:

```
template<class T> class std::valarray
{
  // внутреннее представление

public:
    typedef T value_type;

    valarray();                                 // valarray с size()==0
    explicit valarray(size_t n);                // n элементов со значением T()
    valarray(const T& val, size_t n);           // n элементов со значением val
    valarray(const T* p, size_t n);             // n элементов со значениями p[0], p[1], ...
    valarray(const valarray& v);                // копия v

    valarray(const slice_array<T>&);            // см. §22.4.6
    valarray(const gslice_array<T>&);           // см. §22.4.8
    valarray(const mask_array<T>&);             // см. §22.4.9
    valarray(const indirect_array<T>&);         // см. §22.4.10

    ~valarray();
    // ...
};
```

Данный набор конструкторов позволяет нам инициализировать **valarray**, используя числовые массивы вспомогательных типов или отдельные значения. Например:

```
valarray<double> v0;                    // можно присвоить что-нибудь позже
valarray<float> vl(1000);               // 1000 элементов со значением float()==0.0F
valarray<int> v2(-1,2000);              // 2000 элементов со значением -1
valarray<double> v3(100,9.8064);        // ошибка: размер задан числом с плав. запятой
valarray<double> v4 = v3;               // v4 содержит v3.size() элементов
```

В двухаргументных конструкторах значение указывается перед числом элементов. Это отличается от порядка следования аргументов для стандартных контейнеров (§16.3.4).

Число элементов в качестве аргумента конструкторов определяет итоговый размер **valarray**.

Большинству программ нужны данные из таблиц или из ввода; это поддерживается конструктором, копирующим элементы массивов встроенных типов. Например:

```
const double vd[] = {0,1,2,3,4};
const int vi[] ={0,1,2,3,4};

valarray<double> v3(vd,4);   // 4 элемента: 0,1,2,3
valarray<double> v4(vi,4);   // type error: vi - это не указатель на double
valarray<double> v5(vd,8);   // undefined: слишком мало элементов для инициализации
```

Эта форма инициализации крайне важна, поскольку вычислительные программы часто выдают данные в виде больших массивов.

Тип **valarray** и вспомогательные средства, связанные с этим типом, разработаны для высокопроизводительных вычислений. Это отражается в ряде ограничений для пользователей и в определенной свободе для разработчиков, которым разрешено использовать все мыслимые способы оптимизации. Кроме того, операции с **valarray** можно делать встраиваемыми, они не должны иметь побочных эффектов и не должны иметь синонимов. При сохранении базовой семантики допускаются введение вспомогательных типов и устранение временных объектов. В итоге, объ-

явления в заголовочном файле <*valarray*> могут выглядеть не совсем так, как я представляю их здесь (и в стандарте), но они должны предоставлять те же самые операции с тем же самым смыслом и ожидаемым результатом для любого клиентского кода, не нарушающего установленных правил. В частности, элементы массива valarray должны обладать обычной семантикой копирования (§17.1.4).

22.4.2. Индексирование и присваивание в классе valarray

В классе ***valarray*** операция индексирования применяется как для доступа к отдельным элементам, так и для того, чтобы получать подмассивы:

```
template<class T> class valarray
{
public:
  // ...
  valarray& operator= (const valarray& v) ;             // копирование v
  valarray& operator= (const T& val) ;                  // присвоить val каждому эл-ту

  T operator[] (size_t) const;
  T& operator[] (size_t) ;

  valarray operator[] (slice) const;                    // см. §22.4.6
  slice_array<T> operator[] (slice) ;

  valarray operator[] (const gslice&) const;            // см. §22.4.8
  gslice_array<T> operator[] (const gslice&) ;

  valarray operator[] (const valarray<bool>&) const;    // см. §22.4.9
  mask_array<T> operator[] (const valarray<bool>&) ;

  valarray operator[] (const valarray<size_t>&) const;  // см. §22.4.10
  indirect_array<T> operator[] (const valarray<size_t>&) ;

  valarray& operator= (const slice_array<T>&) ;         // см. §22.4.6
  valarray& operator= (const gslice_array<T>&) ;        // см. §22.4.8
  valarray& operator= (const mask_array<T>&) ;          // см. §22.4.9
  valarray& operator= (const indirect_array<T>&) ;      // см. §22.4.10
  // ...
};
```

Массив ***valarray*** можно присвоить другому массиву этого же типа и размера. С очевидностью ***v1=v2*** копирует каждый элемент ***v2*** в соответствующую позицию массива ***v1***. В случае разных размеров результат такого копирования не определен, а из-за того, что реализация ***valarray*** сильно оптимизирована по скорости работы, не следует ожидать в этом случае какого-либо информативного исключения или иной разумной формы реакции на ошибку.

Дополнительно к рассмотренному общепринятому присваиванию допускается также присваивание скалярной величины. Например, ***v****=7* присваивает каждому элементу ***v*** значение *7*.

Индексация целыми числами вполне традиционна и при этом никакого контроля выхода за допустимые границы индексов не выполняется.

Кроме извлечения индивидуальных элементов операция индексации для ***valarray*** предоставляет четыре способа извлечения подмассивов (§22.4.6). Операции присваивания (и конструкторы — §22.4.1) принимают такие подмассивы в качестве операн-

дов. Подходящий набор операций присваивания для типа ***valarray*** делает ненужным перед выполнением присваивания преобразование вспомогательных типов, таких как ***slice_array***, в тип ***valarray***. Любая реализация может аналогичным образом определить для типа ***valarray*** другие эффективные векторные операции, например, операции + и *. Наконец, существуют известные приемы оптимизации для векторных операций, включающих срезы (slices) и иные вспомогательные векторные типы.

22.4.3. Функции-члены

Для класса ***valarray*** определено некоторое количество совершенно очевидных и несколько менее очевидных функций-членов:

```
template<class T> class valarray
{
public:
// ...
  valarray& operator*= (const T& arg) ;     // v[i]*=arg для каждого элемента
  // аналогично: /=, %=, +=, -=, '=, &=, |=, <<=, и >>=

  T sum () const;                           // сумма элементов (+= для сложения)
  T min () const;                           // наименьшее значение ( < для сравнения)
  T max () const;                           // наибольшее значение ( < для сравнения)

  valarray shift (int i) const;             // логический сдвиг (влево - i>0; вправо - i<0)
  valarray cshift (int i) const;            // циклический сдвиг (влево - i>0; вправо - i<0)

  valarray apply (T f(T) ) const;           // result[i] = f(v[i]) для каждого элемента
  valarray apply (T f(constT&) ) const;

  valarray operator- () const;              // result[i] = -v[i] для каждого элемента
                                            // аналогично: +, ~, !
  valarray<bool> operator! () const;        // result[i] = !v[i] для каждого элемента

  size_t size () const;                          // число элементов
  void resize (size_t n, const T& val = T () ) ; // n элементов со значением val
} ;
```

Если ***size*** () ==***0***, то возвраты ***sum*** (), ***min*** () и ***max*** () не определены.

Далее, если ***v*** — это массив типа ***valarray***, то его можно масштабировать следующим образом: ***v*=.2***, или ***v/=1.3***. То есть применение к вектору скаляра означает его применение к каждому элементу вектора. Обычно легче оптимизировать операцию *=, чем комбинацию отдельных операций * и = (§11.3.1).

Заметьте, что операции, отличные от присваивания, конструируют новый ***valarray***. Например:

```
double incr (double d) {return d+1; }

void f (valarray<double>& v)
{
  valarray<double> v2 = v.apply (incr) ;  // получаем инкрементированный valarray
}
```

Здесь значение ***v*** не изменяется. Функция ***apply*** () не принимает в качестве аргумента (§22.9[1]), к сожалению, объекты функциональных классов (§18.4).

Функции логического и циклического сдвигов, ***shift***() и ***cshift***(), возвращают новый объект типа ***valarray*** с соответствующим образом сдвинутыми элементами, а исходный ***valarray*** оставляют неизменным. Например, циклический сдвиг ***v2=v.cshift(n)*** породит ***v2*** таким образом, что ***v2[i]==v[(i+n)%v.size()]***. Логический сдвиг ***v3=v.shift(n)*** породит ***v3*** таким образом, что ***v3[i]*** равно ***v[i+n]***, если ***i+n*** — допустимый индекс для ***v***, а в противном случае элемент принимает значение по умолчанию. Считается, что ***shift***() и ***cshift***() осуществляют сдвиг влево, если их аргумент положительный, и сдвиг вправо при отрицательном аргументе. Например:

```
void f()
{
  int alpha[] = {1,2,3,4,5,6,7,8};

  valarray<int> v(alpha,8);           // 1,2,3,4,5,6,7,8
  valarray<int> v2 = v.shift(2);      // 3,4,5,6,7,8,0,0
  valarray<int> v3 = v<<2;            // 4,8,12,16,20,24,28,32
  valarray<int> v4 = v. shift(-2);    // 0,0,1,2,3,4,5,6
  valarray<int> v5 = v>>2;            // 0,0,0,1,1,1,1,2
  valarray<int> v6 = v.cshift(2);     // 3,4,5,6,7,8,1,2
  valarray<int> v7 = v. cshift(-2);   // 7,8,1,2,3,4,5,6
}
```

Для массивов ***valarray*** операции >> и << являются операциями битовых сдвигов, но не операциями сдвига в элементах массива и не операциями ввода/вывода (§22.4.4). Для выполнения битовых сдвигов в элементах интегрального типа массива ***valarray*** можно использовать <<= и >>=. Например:

```
void f(valarray<int> vi, valarray<double> vd)
{
  vi <<= 2;   // vi[i]<<=2 для всех элементов vi
  vd <<= 2;   // error: сдвиг не определен для значений с плавающей запятой
}
```

Размеры массивов ***valarray*** изменять можно. Однако ***resize***() — это не та операция, что превращает ***valarray*** в структуру, способную динамически расти как ***vector*** или ***string***. Вместо этого, ***resize***() реинициализирует массив, заменяя текущее содержимое умолчательными значениями. Старое содержимое при этом теряется.

Часто массив ***valarray*** становится после выполнения операции ***resize***() тем, что создается как пустой вектор. Рассмотрим, как мы могли бы инициализировать ***valarray*** из ввода:

```
void f()
{
  int n = 0;
  cin >> n;                        // читаем размер массива
  if(n<=0) error("неверные границы массива");

  valarray<double> v(n);           // создаем массив нужного размера
  int i = 0;
  while(i<n && cin>>v[i++]);       // заполняем массив
  if(i!=n) error("введено слишком мало элементов");
  // ...
}
```

Если ввод нужно обрабатывать в отдельной функции, можно поступить так:

```
void initialize_from_input(valarray<double>& v)
{
  int n = 0;
  cin >> n;                          // читаем размер массива
  if(n<=0) error("неверные границы массива");

  v.resize(n);                       // задаем v правильный размер
  int i = 0;
  while(i<n && cin>>v[i++]);         // заполняем массив
  if(i!=n) error("введено слишком мало элементов");
}

void g()
{
  valarray<double> v;                // массив по умолчанию
  initialize_from_input(v);          // придаем v размер и элементы
  // ...
}
```

Это позволяет избежать копирования большого объема данных.

Если же мы захотим, чтобы массив ***valarray*** с ценными данными рос динамически, нужно обратиться к временному хранилищу:

```
void grow(valarray<int>& v, size_t n)
{
  if(n<=v.size()) return;

  valarray<int> tmp(n);                          // n умолчательных элементов

  Copy(&v[0], &v[v.size()], &tmp[0]);  // копирующий алгоритм из §18.6.1
  v.resize(n);
  copy(&tmp[0], &tmp[v.size()], &v[0]);
}
```

Но это не основное предназначение массивов ***valarray***. Тип ***valarray*** рассчитан на то, что после инициализации размер массива этого типа не меняется.

Элементы массива ***valarray*** формируют последовательность, так что ***v[0]..v[n-1]*** располагаются в памяти друг за другом. Это означает, что для ***valarray<T>*** ***T**** является итератором произвольного доступа (§19.2.1), и можно использовать стандартные алгоритмы, такие как ***copy()***, например. Однако лучше в духе ***valarray*** применять для копирования операцию присваивания и подмассивы:

```
void grow2(valarray<int>& v, size_t n)
{
  if(n<=v.size()) return;

  valarray<int> tmp = v;
  slice s(0,v.size(),1);   // подмассив из v.size() элементов (см. §22.4.5)

  v.resize(n);             // resize() не сохраняет значения элементов
  v[s] = tmp;              // копируем элементы назад в начальную часть v
}
```

Если по какой-то причине ввод данных организован так, что узнать число вводимых данных можно лишь пересчитав их, то лучше сначала ввести данные в кон-

тейнер типа ***vector*** (§16.3.5), и только потом копировать элементы в массив ***valarray***.

22.4.4. Внешние операции и функции

Обычные бинарные операции и математические функции определены для массивов ***valarray***:

```
template<class T> valarray<T> operator* (const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator* (const valarray<T>&, const T&);
template<class T> valarray<T> operator* (const T&, const valarray<T>&);
// аналогично: /,%, +, -, ^,&, |, <<,>>, &&, ||, ==,!=,<, >, <=,>=, atan2 и pow

template<class T> valarray<T> abs (const valarray<T>&);
// аналогично: acos, asin, atan, cos, cosh, exp, log, log10, sin, sinh,s qrt, tan и tanh
```

Бинарные операции определяются для двух операндов ***valarray***, а также для ***valarray*** и его скалярного типа. Например:

```
void f(valarray<double>& v, valarray<double>& v2, double d)
{
  valarray<double> v3 = v*v2;      // v3[i] = v[i]*v2[i] для всех i
  valarray<double> v4 = v*d;       // v4[i] = v[i]*d для всех i
  valarray<double> v5 = d*v2;      // v5[i] = d*v2[i] для всех i

  valarray<double> v6 = cos(v);    // v6[i] = cos(v[i]) для всех i
}
```

Все эти операции применяются ко всем элементам операндов (операнда) способом, показанным в примере для * и ***cos***(). Естественно, операция применима лишь в том случае, когда она имеет смысл для типа аргумента шаблона. В противном случае компилятор выдаст ошибку в момент конкретизации шаблона (§13.5).

Там, где результат имеет тип ***valarray***, длина получающегося массива та же, что и у операндов типа ***valarray***. В случае разной длины операндов ***valarray*** результат бинарных операций не определен.

Довольно любопытно, но для ***valarray*** не предоставляется никаких операций ввода/вывода (§22.4.3); << и >> являются операциями сдвига. В то же время, можно легко определить версии операций << и >> для ввода/вывода (§22.9[5]).

Заметьте, что рассматриваемые операции возвращают новый ***valarray***, а не модифицируют свои операнды. Это может дорого стоить, но может и не стоить, если для реализации ***valarray*** применяется агрессивная оптимизация (см., например, §22.4.7).

Все эти операции и математические функции над массивами ***valarray*** можно применять и к ***slice_array*** (§22.4.7), ***gslice_array*** (§22.4.8), ***mask_array*** (§22.4.9), ***indirect_array*** (§22.4.10) и к комбинациям этих типов. В то же время, реализации могут преобразовывать эти типы в ***valarray*** перед выполнением операций.

22.4.5. Срезы

Срез (***slice***) — это абстракция, которая позволяет эффективно *работать с векторами как с матрицами произвольной размерности*. Это ключевое понятие для векторов языка Fortran и библиотеки BLAS (Basic Linear Algebra Subprograms), являющейся базисом для многих численных расчетов. В своей основе срез — это каждый ***n***-ый элемент некоторой части ***valarray***:

```
class std::slice
{
  // начальный индекс, длина и шаг
public:
  slice();
  slice(size_t start, size_t size, size_t stride);

  size_t start() const;    // индекс первого элемента
  size_t size() const;     // число элементов
  size_t stride() const;   // n-ый элемент находится по
                           // адресу: start()+n*stride()
};
```

Здесь *шаг* (*stride*) — это расстояние (в элементах) между последовательными элементами среза. Таким образом, срез определяется последовательностью целых чисел. Например, следующий код

```
size_t slice_index(const slice& s, size_t i)
{
  return s.start() + i*s.stride();
}

void print_seq(const slice& s)  // вывод элементов среза s
{
  for(size_t i = 0; i<s.size(); i++) cout << slice_index(s,i) << " ";
}

void f()
{
  print_seq(slice(0,3,4));   // строка 0
  cout << ", ";
  print_seq(slice(1,3,4));   // строка 1
  cout << ", ";
  print_seq(slice(0,4,1));   // столбец 0
  cout<< ", ";
  print_seq(slice(4,4,1));   // столбец 1
}
```

выведет

```
0 4 8 , 1 5 9 , 0 1 2 3 , 4 5 6 7.
```

Другими словами, срез описывает отображение неотрицательных целых в индексы. Число элементов (**size()**) не влияет на отображение, а просто позволяет нам найти конец последовательности. Такое отображение позволяет *имитировать двумерные массивы внутри одномерного массива* (такого как **valarray**) эффективно, универсально и относительно просто (удобно). Рассмотрим матрицу 3 на 4 так, как мы часто ее себе представляем (§C.7):

00	01	02
10	11	12
20	21	22
30	31	32

По правилам языка Fortran она располагается в памяти следующим образом:

0				4				8			
0 0	1 0	2 0	3 0	0 1	1 1	2 1	3 1	0 2	1 2	2 2	3 2
0	1	2	3								

В С++ массивы располагаются в памяти иначе (§С.7). Однако нам нужно представить концепцию в рамках ясного и логического интерфейса, а затем уж учитывать ограничения, налагаемые конкретным представлением. Я выбрал расположение в стиле языка Fortran для того, чтобы упростить взаимодействие с программами численных расчетов, следующими этому соглашению. Однако я все же не зашел так далеко, чтобы начинать индексацию с *единицы*, а не с *нуля*; это оставляется в качестве упражнения (§22.9[9]). Многие вычисления выполняются и будут выполняться с применением разных языков программирования и множества различных библиотек. А потому часто важно иметь возможность манипулировать данными в разных форматах, соответствующих этим языкам и библиотекам.

Строку с номером ***x*** можно представить срезом ***slice(x, 3, 4)***. То есть первым элементом строки ***x*** является ***x***-ый элемент вектора, следующий элемент этой строки есть ***(x+4)***-ый элемент вектора и т.д., и в каждой строке всего ***3*** элемента. В соответствии с рисунками ***slice(0, 3, 4)*** описывает нулевую строку с ***00***, ***01*** и ***02***.

Столбец ***y*** можно представить срезом ***slice(4*y, 4, 1)***. То есть первым элементом столбца с номером у является ***4*y***-ый элемент вектора, следующим элементом этого столбца — ***(4*y+1)***-ый элемент вектора и т.д., и в каждом столбце всего ***4*** элемента. В соответствии с рисунками ***slice(0, 4, 1)*** описывает нулевой столбец с ***00***, ***10***, ***20*** и ***30***.

Кроме моделирования двумерных массивов срезы можно использовать для описания и других последовательностей. Они являются довольно универсальным средством представления простых последовательностей. Эта идея подробнее рассматривается в §22.4.8.

Еще можно рассматривать срезы как необычный вид итераторов, ведь они позволяют задать последовательности индексов для ***valarray***. На этой основе можно построить настоящий итератор:

```
template<class T> class Slice_iter
{
  valarray<T>* v;
  slice s;
  size_t curr;       // индекс текущего элемента

  T& ref(size_t i) const {return (*v)[s.start()+i*s.stride()];}
```

```
public:
  Slice_iter(valarray<T>* vv, slice ss) : v(vv), s(ss), curr(0) { }

  Slice_iter end() const
  {
    Slice_iter t = *this;
    t.curr = s.size();                    // индекс эл-та, следующего за последним
    return t;
  }

  Slice_iter& operator++() {curr++; return *this; }
  Slice_iter operator++(int) {Slice_iter t= *this; curr++; return t; }

  T& operator[](size_t i) {return ref(i); }  // индексация в стиле C
  T& operator()(size_t i) {return ref(i); }  // индексация в стиле Fortran
  T& operator*() {return ref(curr); }        // текущий элемент

  friend bool operator==(const Slice_iter& p, const Slice_iter& q);
  friend bool operator!=(const Slice_iter& p, const Slice_iter& q);
  friend bool operator<(const Slice_iter& p, const Slice_iter& q);
};
```

Поскольку срез имеет размер, мы даже можем обеспечить проверку корректности диапазона элементов. Я воспользовался ***slice*::*size*()**, чтобы ввести операцию ***end*()**, предоставляющую итератор, настроенный на элемент, следующий за последним элементом среза.

Поскольку срез может описывать либо строку, либо столбец, ***Slice_iter*** позволяет проходить ***valarray*** как по строкам, так и по столбцам.

Сравнения можно определить следующим образом:

```
template<class T> bool operator==(const Slice_iter<T>& p, const Slice_iter<T>& q)
{
  return p.curr==q.curr && p.s.stride()==q.s.stride() && p.s.start()==q.s.start();
}

template<class T> bool operator!=(const Slice_iter<T>& p, const Slice_iter<T>& q)
{
  return !(p==q);
}

template<class T> bool operator<(const Slice_iter<T>& p, const Slice_iter<T>& q)
{
  return p.curr<q.curr && p.s.stride()==q.s.stride() && p.s.start()==q.s.start();
}
```

22.4.6. Массив slice_array

Из массива ***valarray*** и среза мы можем построить нечто похожее на ***valarray***, но на самом деле являющееся лишь способом обратиться к подмножеству массива, адресуемого срезом. Это ***slice_array***, который определяется следующим образом:

```
template<class T> class std::slice_array
{
public:
  typedef T value_type;
```

```
    void operator= (const valarray<T>&) ;
    void operator= (const T& val) ;               // присвоить val каждому элементу

    void operator*= (const valarray<T>& val) ; // v[i]*=val[i] для каждого элемента
    // аналогично: /=, %=, +=, -=, ^=, &=, |=, <<=, >>=

    ~slice_array () ;

private:
    slice_array () ;                               // предотвращаем конструирование
    slice_array (const slice_array&) ;             // предотвращаем копирование
    slice_array& operator= (const slice_array&) ;  // предотвращаем копирование

    valarray<T>* p;                                // зависящее от реализации представление
    slice s;
};
```

Пользователь не может напрямую создавать объекты типа ***slice_array***. Вместо этого пользователь индексирует ***valarray***, чтобы создать ***slice_array*** для данного среза. Как только ***slice_array*** инициализирован, все обращения к нему косвенно передаются ***valarray***, для которого он создан. Например, мы можем создать нечто, что представляет каждый второй элемент массива следующим образом:

```
void f (valarray<double>& d)
{
    slice_array<double>& v_even = d[slice (0, d.size () /2+d.size () %2, 2) ] ;
    slice_array<double>& v_odd  = d[slice (1, d.size () /2, 2) ] ;

    v_even *= v_odd;    // перемножить элементы попарно и сохранить
                        // результат в четных элементах
    v_odd = 0;          // присвоить 0 каждому нечетному элементу d
}
```

Запрет на копирование ***slice_array*** необходим, чтобы поддержать оптимизации, рассчитывающие на отсутствие синонимов объектов. Это может оказаться стеснительным. Например:

```
slice_array<double> row (valarray<double>& d, int i)
{
    slice_array<double> v = d[slice (0, 2, d.size () /2) ] ; // error: попытка копирования
    return d[slice (i%2, i, d.size () /2) ] ;                // error: попытка копирования
}
```

Часто копирование срезов является разумной альтернативой копирования ***slice_array***.

Срезы могут использоваться для формирования разных подмножеств массивов. Например, можно применить срезы для манипулирования непрерывно расположенными подмассивами следующим образом:

```
inline slice sub_array (size_t first, size_t count)      // [first:first+count[
{
    return slice (first, count, 1) ;
}

void f (valarray<double>& v)
{
```

```
    size_t sz = v.size();
    if(sz<2) return;
    size_t n = sz/2;
    size_t n2 = sz-n;

    valarray<double> half1(n);
    valarray<double> half2(n2);

    half1 = v[sub_array(0,n)];      // копирование первой половины v
    half2 = v[sub_array(n,n2)];     // копирование второй половины v
    // ...
}
```

Стандартная библиотека не предоставляет никаких матричных классов. Вместо этого, она предлагает использовать ***valarray*** и срезы в качестве инструмента построения матриц, оптимизированных для разных задач. Рассмотрим, как можно при помощи ***valarray*** и ***slice_array*** реализовать простую двумерную матрицу:

```
class Matrix
{
  valarray<double>* v;              // хранит эл-ты поколонно, как описано в §22.4.5
  size_t d1, d2;                    // d1 - кол-во столбцов, d2 - кол-во строк

public:
  Matrix(size_t x, size_t y);       // внимание: умолчательного конструктора нет
  Matrix(const Matrix&);
  Matrix& operator=(const Matrix&);
  ~Matrix() {delete v;}

  size_t size() const {return d1*d2;}
  size_t dim1() const {return d1;}  // кол-во столбцов
  size_t dim2() const {return d2;}  // кол-во строк

  Slice_iter<double> row(size_t i);
  Cslice_iter<double> row(size_t i) const;
  Slice_iter<double> column(size_t i);
  Cslice_iter<double> column(size_t i) const;

  double& operator()(size_t x, size_t y);           // индексация в стиле Fortran
  double operator()(size_t x, size_t y) const;

  Slice_iter<double> operator()(size_t i) {return column(i);}
  Cslice_iter<double> operator()(size_t i) const {return column(i);}

  Slice_iter<double> operator[](size_t i) {return column(i);}  // индексация в стиле C
  Cslice_iter<double> operator[](size_t i) const {return column(i);}

  Matrix& operator*=(double);

  valarray<double>& array() {return *v;}
};
```

Здесь матричный класс ***Matrix*** построен на базе массива ***valarray***. Размерность мы «добавляем» здесь с помощью срезов. По мере необходимости мы можем рассматривать это представление как одномерное, двумерное, трехмерное и т.д. так же, как мы ввели умолчательное двумерное представление с помощью ***row***() и ***column***(). Здесь

Slice_iter используется для обхода запрета на копирование ***slice_array***. Я не могу возвратить ***slice_array***

```
slice_array<double> row(size_t i) {return (*v)[slice(i,d1,d2)];} // error
```

и поэтому я возвращаю итератор, содержащий указатель на ***valarray*** и собственно срез.

Нам еще требуется тип «указатель на срез констант» — ***Cslice_iter***, чтобы отразить отличие в срезах для константных и неконстантных ***Matrix***:

```
inline Slice_iter<double> Matrix::row(size_t i)
{
  return Slice_iter<double>(v,slice(i,d1,d2));
}

inline Cslice_iter<double> Matrix::row(size_t i) const
{
  return Cslice_iter<double>(v,slice(i,d1,d2));
}

inline Slice_iter<double> Matrix::column(size_t i)
{
  return Slice_iter<double>(v,slice(i*d2,d2,1));
}

inline Cslice_iter<double> Matrix::column(size_t i) const
{
  return Cslice_iter<double>(v,slice(i*d2,d2,1));
}
```

Определение итератора ***Cslice_iter*** идентично определению ***Slice_iter*** за исключением того, что возвращаются константные ссылки на элементы среза.

Остальные функции-члены довольно тривиальны:

```
Matrix::Matrix(size_t x, size_t y)
{
  // проверка осмысленности x и y
  d1 =x;
  d2 =y;
  v = new valarray<double>(x*y);
}

double& Matrix::operator()(size_t x, size_t y)
{
  return column(x)[y];
}

double mul(const Cslice_iter<double>& v1, const valarray<double>& v2)
{
  double res = 0;
  for(size_t i = 0; i<v2.size(); i++) res += v1[i]*v2[i];
  return res;
}

valarray<double> operator*(const Matrix& m, const valarray<double>& v)
{
```

```
    valarray<double> res(m.dim2());
    for(size_t i = 0; i<m.dim2(); i++) res[i] = mul(m.row(i),v);
    return res;
}

Matrix& Matrix::operator*=(double d)
{
    (*v) *= d;
    return *this;
}
```

Для индексации матрицы я использую выражение (*i,j*), поскольку операция () — единственная, и потому что такое выражение прекрасно знакомо всем специалистам-вычислителям. Концепция строки обеспечивает более знакомое (для программистов на C и C++) выражение [*i*][*j*]:

```
void f(Matrix& m)
{
    m(1,2) = 5;          // индексация в стиле Fortran
    m.row(1)(2) = 6;
    m.row(1)[2] = 7;
    m[1](2) = 8;         // нежелательный смешанный стиль (работает)
    m[1][2] = 9;         // индексация в стиле C++
}
```

Использование ***slice_array*** в целях реализации индексации предполагает наличие хорошего оптимизатора кода.

Обобщение этого подхода на случай ***n***-мерных матриц произвольных элементов с разумным набором операций оставляется в качестве упражнения (§22.9[7]).

Возможно, ваша первая идея насчет двумерной матрицы может выглядеть примерно так:

```
class Matrix
{
    valarray<valarray<double> > v;
    // ...
};
```

Однако для типа ***valarray*** гарантируется лишь работа со скалярными элементами, так что нам придется писать что-то вроде

```
vector<vector<double> > v;
```

и совсем не просто достичь эффективности и совместимости, необходимых для высокопроизводительных вычислений, не опускаясь на традиционный низкий уровень, представленный типом ***valarray*** и срезами.

22.4.7. Временные объекты, копирование, циклы

Если вы строите векторный или матричный класс, то скоро обнаружите три проблемы, решения которых от вас ожидают пользователи, которым требуется высокая производительность:

1. Минимизация временных объектов.
2. Минимизация копирования матриц.
3. Минимизация вложенных циклов над одними и теми же данными.

Эти проблемы не связаны напрямую с основной целью стандартной библиотеки. Здесь я только схематически опишу прием, позволяющий осуществить высокооптимизированные реализации.

Рассмотрим ***U=M* V+W***, где ***U***, ***V*** и ***W*** — векторы, а ***M*** — матрица. Наивная реализация предоставит временные векторы для ***M* V*** и ***M* V+W***, и будет копировать результаты вычисления этих выражений. Более искусная реализация введет функцию вроде ***mul_add_and_assign***(&***U***, &***M***, &***V***, &***W***), которая обходится без временных векторов, не копирует то и дело элементы и, вообще, обращается к элементам по минимуму.

Столь глубокая оптимизация требуется лишь для нескольких типов выражений, так что простое решение проблемы состоит в том, чтобы предоставить не слишком сложную функцию вроде ***mul_add_and_assign***(), и пусть пользователь вызывает ее там, где важна эффективность. С другой стороны, можно спроектировать ***Matrix*** таким образом, чтобы для определенного вида выражений оптимизация выполнялась бы автоматически, например, для выражений ***U=M* V+W*** можно применять единственную операцию с четырьмя операндами. Соответствующие приемы ранее были показаны на примере манипуляторов для потоков ostream (§21.4.6.3) — ими можно пользоваться, чтобы комбинация из n бинарных операций работала как одна ***n***-арная операция. Все это может обеспечить резкое (раз в 30) повышение производительности за счет внедрения еще более мощных приемов оптимизации.

Теперь определим результат умножения ***Matrix*** на ***Vector***:

```
struct MVmul
{
  const Matrix& m;
  const Vector& v;

  MVmul(const Matrix& mm, const Vector& vv) : m(mm), v(vv) {}

  operator const Vector();
};

inline MVmul operator*(const Matrix& mm, const Vector& vv)
{
  return MVmul(mm,vv);
}
```

Это «умножение» ничего не делает — лишь хранит ссылки на свои операнды; реальное же вычисление ***M* V*** откладывается. Объект, порождаемый операцией *, соответствует тому, что во многих других областях (и языках программирования) принято называть *замыканием* (*closure*).

Теперь аналогичным образом вводим сложение с вектором:

```
struct MVmulVadd
{
  const Matrix& m;
  const Vector& v;
  const Vector& v2;
```

```
  MVmulVadd(const MVmul& mv, const Vector& vv) : m(mv.m), v(mv.v), v2(vv) { }
  operator const Vector();
};

inline MVmulVadd operator+(const MVmul& mv, const Vector& vv)
{
  return MVmulVadd(mv,vv);
}
```

В результате реальное вычисление *M*V+W* откладывается «на потом», но в конце концов, нам нужно гарантировать реальное эффективное вычисление в момент присвоения этого выражения вектору:

```
void mul_add_and_assign(Vector*, const Matrix*, const Vector*, const Vector*);

class Vector
{
public:
  Vector(const MVmulVadd& m)
  {
    // размещение элементов и т.д.
    mul_add_and_assign(this, &m.m, &m.v, &m.v2);
  }

  Vector& operator=(const MVmulVadd& m)
  {
    mul_add_and_assign(this, &m.m, &m.v, &m.v2);
    return *this;
  }
  // ...
};
```

Выражение *U=M*V+W* автоматически раскрывается в

```
U.operator=(MVmulVadd(MVmul(M,V),W))
```

и что из-за встраивания сводится просто к желаемому вызову

```
mul_add_and_assign(&U, &M, &V, &W)
```

Ясно, что тем самым устраняются копирование и создание временных объектов. Можно, конечно, оптимизировать и саму функцию ***mul_add_and_assign()***, но даже в своей простой и неоптимизированной версии она оставляет массу возможностей программисту-оптимизатору.

Я ввел новый тип ***Vector***, а не использовал ***valarray***, потому что мне нужно было определить операцию присваивания (а она должна определяться как функция-член; §11.2.2). В то же время, ***valarray*** — это наилучшая кандидатура для внутреннего представления типа ***Vector***.

Важность рассмотренного приема заключается в том, что большая часть критически важных по производительности вычислений с матрицами и векторами выполняются с помощью немногочисленных и довольно простых синтаксических конструкций. В типичном случае не возникает нужды в сильной оптимизации дюжин операций — для них достаточно привычных методов (§11.6).

Рассмотренный прием, основанный на опыте практической компиляции, вводит замыкающие объекты, позволяющие переместить реальные вычисления подвыражений в объект, представляющий композитную операцию. Этот подход можно применять ко многим задачам, имеющим следующую общую черту: отдельные части информации нужно собрать в одну функцию до того момента, когда начнутся реальные вычисления. Я называю объекты, позволяющие отложить реальные вычисления, *объектами, замыкающими композицию* (*composition closure objects*).

22.4.8. Обобщенные срезы

Пример с ***Matrix*** в §22.4.6 показывает, как можно использовать два среза для описания строк и столбцов двумерного массива. В общем случае срез может описывать любую строку или столбец ***n***-мерного массива (§22.9[7]). Но иногда нам требуется извлечь подмассив, который не является ни строкой, ни столбцом. Например, нам может потребоваться матрица 2х3 из левого верхнего угла матрицы 3х4:

00	01	02
10	11	12
20	21	22
30	31	32

К сожалению, нужные нам элементы не располагаются таким образом, чтобы их можно было охарактеризовать одним срезом:

0	1	2									
0 0	1 0	2 0	3 0	0 1	1 1	2 1	3 1	0 2	1 2	2 2	3 2
				4	5	6					

Класс ***gslice*** — это *обобщенный срез* (*generalized slice*), который содержит (почти) всю информацию из ***n*** срезов:

```
class std::gslice
{
  // gslice содержит n шагов и n размеров
public:
  gslice();
  gslice(size_t s, const valarray<size_t>& l, const valarray<size_t>& d);

  size_t start() const;                 // индекс первого элемента
  valarray<size_t> size() const;        // число элементов измерения
  valarray<size_t> stride() const;      // шаг для index[0], index{1}, ... .
};
```

Дополнительные значения позволяют ***gslice*** определить соответствие между ***n*** целыми числами и индексом, которым адресуются элементы массива. Напри-

мер, мы можем описать размещение матрицы 2x3 двумя парами (длина, шаг) целых чисел. Как показано в §22.4.5, при расположении элементов в стиле языка Fortran длина *2* и шаг *4* описывают два элемента строки матрицы 3x4. Аналогично, длина *3* и шаг *1* описывают три элемента столбца. В совокупности они описывают элементы подматрицы 2x3. Для перечисления элементов мы можем написать:

```
size_t gslice_index (const gslice& s, size_t i, size_t j)
{
  return s.start() + i*s.stride()[0] + j*s.stride()[1];
}

size_t len[] = {2,3};     // (len[0],str[0]) описывают строку
size_t str[] = {4,1};     // (len[l],str[l]) описывают столбец

valarray<size_t> lengths (len, 2);
valarray<size_t> strides (str, 2);

void f()
{
  gslice s (0, lengths, strides);

  for (int i = 0; i<s.size()[0]; i++) cout << gslice_index(s,i,0) << " "; // строка
  cout<< ", ";
  for (int j = 0; j<s.size()[1]; j++) cout << gslice_index{s,0,j) << " "; // столбец
}
```

При этом будет выведено *0 4 , 0 1 2*.

Таким образом, ***gslice*** с двумя парами (длина, шаг) описывает подмассив двумерного массива, ***gslice*** с тремя парами (длина, шаг) описывает подмассив трехмерного массива, и так далее. Использование ***gslice*** в качестве индекса для ***valarray*** породит ***gslice_array***, состоящий из элементов, которые описывает ***gslice***. Например:

```
void f(valarray<float>& v)
{
  gslice m (0, lengths, strides);
  v[m] = 0;                // обнуляем v[0],v[l],v[2],v[4],v[5],v[6]
}
```

Массив ***gslice_array*** предоставляет тот же набор членов, что и ***slice_array***. В частности, пользователь не может напрямую конструировать ***gslice_array***, а также его нельзя копировать (§22.4.6). Объекты ***gslice_array*** порождаются в качестве результата использования ***gslice*** для индексации элементов ***valarray*** (§22.4.2).

22.4.9. Маски (тип mask_array)

Массив ***mask_array*** (маска) предоставляет еще один способ определения подмножества ***valarray*** и предоставления результата в виде, похожем на ***valarray***. В контексте массивов ***valarray*** маски — это просто ***valarray<bool>***. Когда маски применяются для индексации ***valarray***, значение ***true*** означает, что элемент включается в подмножество. Это позволяет нам оперировать подмножеством ***valarray*** даже тогда, когда нет четкой структуры (например ***slice***), описывающей подмножество. Например:

```
void f(valarray<double>& v)
{
  bool b[] = {true,false,false,true,false,true};
  valarray<bool> mask(b,6);              // элементы 0, 3 и 5
  valarray<double> vv = cos(v[mask]);    // vv[0]==cos(v[0]), vv[1]==(cos(v[3]),
                                         // vv[2]==cos(v[5])
}
```

Маски предоставляют тот же набор членов, что и тип ***slice_array***. В частности, пользователь не может напрямую конструировать ***mask_array***, а также их нельзя копировать (§22.4.6). Объекты ***mask_array*** порождаются в качестве результата использования ***valarray<bool>*** для индексации элементов ***valarray*** (§22.4.2). Число элементов массива ***valarray***, используемого в качестве маски, не должно превышать число элементов индексируемого таким образом массива ***valarray***.

22.4.10. Тип indirect_array

Тип ***indirect_array*** предоставляет способ произвольного выделения подмножеств из ***valarray*** и его переупорядочения. Например:

```
void f(valarray<double>& v)
{
  size_t i[] = {3,2,1,0};              // первые четыре элемента в обратном порядке
  valarray<size_t> index(i,4);         // элементы 3,2,1,0
  valarray<double> vv = log(v[index]); // vv[0]==log(v[3]), vv[l]==log(v[2]),
                                       // vv[2]==log(v[l]), vv[3]==log(v[0])
}
```

Если индекс указан дважды, значит мы сослались на соответствующий элемент ***valarray*** дважды в одной операции. Это соответствует синонимии, которую ***valarray*** запрещает, так что если индекс повторяется, то результирующий ***indirect_array*** не определен.

Тип ***indirect_array*** предоставляют тот же набор членов, что и тип ***slice_array***. В частности, пользователь не может напрямую конструировать объекты ***indirect_array***, а также их нельзя копировать (§22.4.6). Объекты ***indirect_array*** порождаются в качестве результата использования ***valarray<size_t>*** для индексации элементов ***valarray*** (§22.4.2). Число элементов массива ***valarray***, используемого для индексации, не должно превышать число элементов индексируемого таким образом массива ***valarray***.

22.5. Комплексная арифметика

Стандартная библиотека предлагает шаблон ***complex***, реализующий комплексную арифметику в духе класса ***complex***, рассмотренного в §11.3. Библиотечный ***complex*** должен быть шаблоном, чтобы комплексные числа могли базироваться на разных скалярных типах. В частности, представлены специализации ***complex*** для ***float***, ***double*** и ***long double*** в качестве его скалярных типов.

Шаблон ***complex*** определяется в пространстве имен ***std*** и представлен в заголовочном файле <***complex***>:

```
template<class T> class std::complex
{
  T re, im;
public:
  typedef T value_type;

  complex (const T& r = T() , const T& i = T() ) : re(r) , im(i) { }
  template<class X> complex (const complex<X>& a) : re(a.real() ) , im(a.imag() ) { }

  T real() const {return re;}
  T imag() const {return im;}

  complex<T>& operator= (const T& z) ; // присвоить complex(z,0)
  template<class X> complex<T>& operator= (const complex<X>&) ;
  // аналогично: +=, -=, *=, /=
};
```

Данное представление и встраиваемые функции приведены для иллюстрации. Трудно, однако, представить что-либо иное для стандартного библиотечного ***complex***. Обратите здесь внимание на шаблоны в качестве членов ***complex*** — они обеспечивают инициализацию и присваивание комплексных чисел с другим скалярным типом (§13.6.2).

На протяжении всей книги я использовал ***complex*** в качестве класса, а не шаблона. Это допустимо из-за небольшого фокуса с пространством имен и оператором ***typedef***:

```
typedef std::complex<double> complex;
```

Для complex определяются обычные унарные и бинарные операции:

```
template<class T> complex<T> operator+ (const complex<T>&, const complex<T>&) ;
template<class T> complex<T> operator+ (const complex<T>&, const T&) ;
template<class T> complex<T> operator+ (const T&, const complex<T>&) ;

// аналогично: -, *, I, == и !=

template<class T> complex<T> operator+ (const complex<T>&) ;
template<class T> complex<T> operator- (const complex<T>&) ;
```

Имеются координатные функции:

```
template<class T> T real (const complex<T>&) ;
template<class T> T imag (const complex<T>&) ;

template<class T> complex<T> conj (const complex<T>&) ;

// конструируется из полярных координат (abs(),arg()):
template<class T> complex<T> polar (const T& rho, const T& theta) ;

template<class T> T abs (const complex<T>&) ;
template<class T> T arg (const complex<T>&) ;

template<class T> T norm (const complex<T>&) ;      // квадрат abs()
```

Предоставляются обычные математические функции:

```
template<class T> complex<T> sin (const complex<T>&) ;
// аналогично: sinh, sqrt, tan, tanh, cos, cosh, exp, log и log10

template<class T> complex<T> pow (const complex<T>&, int) ;
template<class T> complex<T> pow (const complex<T>&, const T&) ;
template<class T> complex<T> pow (const complex<T>&, const complex<T>&) ;
template<class T> complex<T> pow (const T&, const complex<T>&) ;
```

И, наконец, обеспечивается потоковый ввод/вывод:

```
template<class T, class Ch, class Tr>
basic_istream<Ch,Tr>& operator>> (basic_istream<Ch,Tr>&, complex<T>&) ;
template<class T, class Ch, class Tr>
basic_ostream<Ch,Tr>& operator<< (basic_ostream<Ch,Tr>&, const complex<T>&) ;
```

Комплексные числа выводятся в формате (*x*, *y*), и могут читаться в форматах *x*, (*x*) и (*x*, *y*) (§21.2.3, §21.3.5). Специализации **complex<float>**, **complex<double>** и **complex<long double>** введены для ограничения необходимости в преобразованиях типа (§13.6.2) и для оптимизации. Например:

```
template<> class complex<double>
{
  double re, im;

public:
  typedef double value_type;

  complex (double r = 0.0, double i = 0.0) : re (r) , im (i) { }
  complex (const complex<float>& a) : re (a.real () ) , im (a.imag () ) { }
  explicit complex (const complex<long double>& a) : re (a.real () ) , im (a.imag () ) { }
  // ...
};
```

Теперь **complex<float>** может «тихо» преобразовываться в **complex<double>**, в то время как **complex<long double>** — нет. Аналогично, специализации гарантируют неявные преобразования **complex<float>** и **complex<double>** в **complex<long double>**, а обратные преобразования — нет. Забавно, что присваивания подобной защиты не предоставляют. Например:

```
void f(complex<float> cf, complex<double> cd, complex<long double> cdl,
complex<int> ci)
{
  complex<double> c1 = cf;       // чудесно
  complex<double> c2 = cd;       // чудесно
  complex<double> c3 = cld;      // error: возможно усечение
  complex<double> c4 (cld) ;     // ok: явное преобразование
  complex<double> c5 = ci;       // error: нет преобразования

  c1 = cld;    // ok, но будьте осторожны
  c1 = cf;     // ok
  c1 = ci;     // ok
}
```

22.6. Обобщенные численные алгоритмы

В заголовочном файле <***numeric***> стандартная библиотека предоставляет ряд обобщенных численных алгоритмов в стиле нечисленных алгоритмов из <***algorithm***> (глава 18):

Обобщенные численные алгоритмы <numeric>	
accumulate ()	Аккумулирует результаты операции над последовательностью
inner_product ()	Аккумулирует результаты операции над двумя последовательностями
partial_sum ()	Генерирует последовательность по операции над последовательностью
adjacent_difference ()	Генерирует последовательность по операции над последовательностью

Эти алгоритмы обобщают обычные операции, например, вычисление суммы, позволяя применять их ко всем видам последовательностей и позволяя задавать в виде параметра операцию, применимую к их элементам. Каждый алгоритм сопровождается версией, применяющей наиболее типичную для данного алгоритма операцию.

22.6.1. Алгоритм accumulate()

Алгоритм ***accumulate*** () есть обобщение суммирования элементов вектора. Он определяется в пространстве имен ***std*** и представлен в заголовочном файле <***numeric***>:

```
template<class In, class T> T accumulate (In first, In last, T init)
{
  while (first != last) init = init + *first++;
  return init;
}

template<class In, class T, class BinOp>
T accumulate (In first, In last, T init, BinOp op)
{
  while (first != last) init = op (init, *first++) ;
  return init;
  // ...
}
```

Простая версия ***accumulate*** () суммирует элементы последовательности, используя операцию + для ее элементов. Например:

```
void f(vector<int>& price, list<float>& incr)
{
  int i = accumulate (price.begin () ,price.end () , 0) ;
  double d = 0;
```

```
  d = accumulate(incr.begin(),incr.end(),d);
  //...
}
```

Обратите внимание на то, как тип переданного начального значения определяет тип возвращаемого значения.

Не все элементы, которые мы хотим просуммировать, изначально доступны как элементы последовательности. В таком случае мы можем предоставить операцию, которая вызывалась бы из ***accumulate***() для создания добавляемых элементов. Чаще всего это требуется в случае, когда нужно извлекать добавляемые (суммируемые) значения из некоторой структуры данных. Например:

```
struct Record
{
  //...
  int unit_price;
  int number_of_units;
};

long price(long val, const Record& r)
{
  return val + r.unit_price * r.number_of_units;
}

void f(const vector<Record>& v)
{
  cout << "Total value: " << accumulate(v.begin(),v.end(),0,price) << '\n';
}
```

22.6.2. Алгоритм inner_product()

Накопление из последовательности весьма распространенная задача. Но накопление из двух последовательностей тоже встречается не слишком редко. Алгоритм ***inner_product***() определен в пространстве имен ***std*** и представлен в заголовочном файле <***numeric***>:

```
template<class In, class In2, class T>
T inner_product(In first, In last, In2 first2, T init)
{
  while(first != last) init = init + *first++ * *first2++;
  return init;
}

template<class In, class In2, class T, class BinOp, class BinOp2>
T inner_product(In first, In last, In2 first2, T init, BinOp op, BinOp2 op2)
{
  while(first != last) init = op(init,op2(*first++,*first2++));
  return init;
}
```

Как обычно, в качестве аргумента передается только начало второй последовательности. Считается, что вторая последовательность не короче первой последовательности.

Алгоритм ***inner_product()*** является ключевой операцией при умножении матрицы ***Matrix*** на вектор ***valarray***:

```
valarray<double> operator* (const Matrix& m, valarray<double>& v)
{
  valarray<double> res(m.dim2());

  for (size_t i = 0; i<m.dim2(); i++)
  {
    const Cslice_iter<double>& ri = m.row(i);
    res[i] = inner_product(ri,ri.end(),&v[0],double(0));
  }
  return res;
}

valarray<double> operator* (valarray<double>& v, const Matrix& m)
{
  valarray<double> res(m.dim1());

  for (size_t i = 0; i<m.dim1(); i++)
  {
    const Cslice_iter<double>& ci = m.column(i);
    res[i] = inner_product(ci,ci.end(),&v[0],double(0));
  }
  return res;
}
```

Некоторые формы алгоритма ***inner_product()*** называют *скалярным произведением* (*dot product*).

22.6.3. Приращения (incremental changes)

Алгоритмы ***partial_sum()*** и ***adjacent_difference()*** обратны друг другу и работают с приращениями (а не с абсолютными значениями элементов). Они определены в пространстве имен ***std*** и представлены в заголовочном файле <***numeric***>:

```
template<class In, class Out> Out adjacent_difference(In first, In last, Out res);

template<class In, class Out, class BinOp>
  Out adjacent_difference(In first, In last, Out res, BinOp op);
```

Для последовательности ***a,b,c,d*** и т.д. ***adjacent_difference()*** выдаст ***a, b-a,c-b,d-c*** и т.д.

Рассмотрим вектор температурных показаний. Его можно преобразовать в вектор температурных приращений следующим образом:

```
vector<double> temps;

void f()
{
  adjacent_difference(temps.begin(),temps.end(),temps.begin());
}
```

К примеру, последовательность показаний ***17, 19, 20, 20, 17*** превратится в ***17, 2, 1, 0, -3***.

Алгоритм ***partial_sum()***, наоборот, позволяет получить последовательность значений из последовательности приращений:

```
template<class In, class Out, class BinOp>
Out partial_sum (In first, In last, Out res, BinOp op)
{
  if(first==last) return res;
  *res = *first;

  typename iterator_traits<In>::value_type val =*first;   // см. §19.2.2

  while (++first != last)
  {
    val = op(val, *first);
    *++res = val;
  }
  return ++res;
}

template<class In, class Out>
Out partial_sum (In first, In last, Out res)
{
  return partial_sum(first,last,res,plus);                 // §18.4.3
}
```

Например, из *a*, *b*, *c*, *d* и т.д. *partial_sum*() производит последовательность *a*, *a*+*b*, *a*+*b*+*c*, *a*+*b*+*c*+*d* и т.д. Например:

```
void f()
{
  partial_sum(temps.begin(),temps.end(),temps.begin());
}
```

Обратите внимание на то, как *partial_sum*() инкрементирует res перед присвоением через него нового значения. Это позволяет в качестве выходной последовательности использовать входную последовательность; *partial_sum*() ведет себя аналогично. Так, код

```
partial_sum(v.begin(),v.end(),v.begin());
```

превращает последовательность *a*, *b*, *c*, *d* в *a*, *a*+*b*, *a*+*b*+*c*, *a*+*b*+*c*+*d* и т.д., а код

```
adjacent_difference(v.begin(),v.end(),v.begin());
```

возвратит последовательность в ее начальное состояние. В частности, *partial_sum*() превратит *17*, *2*, *1*, *0*, *-3* в *17*, *19*, *20*, *20*, *17*.

Эти две операции полезны при анализе изменений любых значений. Например, анализ изменений курса акций нуждается именно в этих операциях.

22.7. Случайные числа

Случайные числа очень важны для моделирования и программирования игр. В заголовочных файлах <*cstdlib*> и <*stdlib.h*> стандартная библиотека предоставляет простые базовые средства для генерации случайных чисел:

```
#define RAND_MAX implementation_defined   /* большое положительное число */
```

```
int rand () ;                      // псевдослучайное число между 0 и RAND_MAX
void srand (unsigned int i) ;     // инициализирует генератор случайных чисел числом i
```

Создать хороший генератор случайных чисел не так легко и, к сожалению, не все системы располагают качественной функцией ***rand***(). В частности, младшие биты в случайных числах часто вызывают подозрение, так что ***rand***()%***n*** — это не совсем хороший переносимый способ генерирования случайных от ***0*** до ***n-1***. Часто, однако, приемлемый результат получается в результате вычисления выражения ***int***((***double***(***rand***())/***RAND_MAX***)****n***). В случае серьезных применений этой формулы нужно позаботиться еще о том, чтобы результат равнялся n с минимальнейшей вероятностью.

Вызов функции ***srand***() инициирует новую последовательность случайных чисел из начального числа ***seed*** («из семени»), заданного в качестве аргумента этой функции. Для отладки важно, чтобы последовательность случайных чисел из заданного «семени» была воспроизводимой. В то же время, бывает нужно, чтобы каждый запуск программы использовал разное «семя» для генерации случайных чисел: например, для непредсказуемости хода игры хорошее семя получается из битов таймера реального времени.

Если вам нужно писать свой собственный генератор случайных чисел, настройтесь на его тщательное тестирование (§22.9[14]).

Генератор случайных чисел удобно реализовать в форме класса. В этом случае легко строятся генераторы случайных чисел с разными распределениями:

```
class Randint                     // равномерное распределение; полагаем 32-bit long
{
  unsigned long randx;

public:
  Randint (long s = 0) {randx=s; }
  void seed (long s) {randx=s; }

  // магические числа позволяют использовать 31 бит из 32-битного long:
  static long abs (long x) {return x&0x7fffffff; }
  static double max () {return 2147483648.0; }          // внимание: double
  long draw () {return randx = randx * 1103515245 + 12345; }

  double fdraw () {return abs (draw ()) /max () ; }      // интервал [0,1]
  long operator () () {return abs (draw () ) ; }         // интервал [0,pow(2,31)]
};

class Urand: public Randint    // равномерное распределение в интервале [0:n[
{
  long n;

public:
  Urand (long nn) {n = nn; }
  long operator () () {long r = n*fdraw () ; return (r==n) ? n-1 : r; }
};

class Erand: public Randint  // экспоненциальное распределение случайных чисел
{
  long mean;
```

```
public:
  Erand(long m) {mean=m;}
  long operator()() {return -mean * log((max()-draw())/max() + .5);}
};
```

Вот простой тест:

```
int main()
{
  Urand draw(10);
  map<int,int> bucket;

  for(int i = 0; i< 1000000; i++) bucket[draw()]++;
  for(int j = 0; j<10; j++) cout << bucket[j] << '\n';
}
```

Если значения ***bucket*** не равны приблизительно ***100000***, то в программе где-то закралась ошибка.

Эти генераторы случайных чисел являются слегка подредактированными версиями того, что я включал в первые выпуски библиотеки C++ (фактически, это было «C с классами»; §1.4).

22.8. Советы

1. Численные задачи часто весьма тонки. Если вы не на 100% уверены в математической стороне дела, проконсультируйтесь со специалистом или поэкспериментируйте; §22.1.
2. Используйте ***numeric_limits*** для определения свойств встроенных типов; §22.2.
3. Специализируйте ***numeric_limits*** для скалярных типов, определяемых пользователем; §22.2.
4. Для математических вычислений, в которых быстродействие важнее гибкости по отношению к типам и операциям, используйте ***valarray***; §22.4.
5. Операции над частью массивов выражайте с помощью срезов, а не с помощью циклов; §22.4.6 .
6. Для достижения эффективности (избавления от временных объектов-массивов) применяйте технику композиции нескольких операций в единую функцию; §22.4.7.
7. Для комплексной арифметики применяйте ***std::complex***; §22.5.
8. Можно с помощью оператора ***typedef*** перенастроить старый код, применяющий класс ***complex***, на работу с шаблоном ***std::complex***; §22.5.
9. Рассмотрите возможность применения алгоритмов ***accumulate()***, ***inner_product()***, ***partial_sum()*** и ***adjacent_difference()*** перед тем, как писать циклы вычислений по значениям элементов последовательностей; §22.6.
10. Отдавайте предпочтению классу случайных чисел с конкретным распределением непосредственному вызову функции ***rand()***; §22.7.
11. Убедитесь, что ваши случайные числа действительно случайны; §22.7.

22.9. Упражнения

1. (*1.5) Напишите функцию, которая ведет себя как ***apply***() из §22.4.3, но не является функцией-членом и принимает в качестве аргумента объекты функциональных классов.
2. (*1.5) Напишите функцию, которая ведет себя как ***apply***() из §22.4.3, но не является функцией-членом, принимает в качестве аргумента объекты функциональных классов и изменяет аргумент типа ***valarray***.
3. (*2) Завершите ***Slice_iter*** (§22.4.5). Будьте особо внимательны, определяя деструктор.
4. (*1.5) Перепишите программу из §17.4.1.3, применив ***accumulate***().
5. (*2) Реализуйте операции ввода/вывода << и >> для ***valarray***. Реализуйте функцию ***get_array***(), которая создает массив ***valarray*** по размеру, задаваемому при вводе.
6. (*2.5) Определите и реализуйте трехмерную матрицу с подходящими операциями.
7. (*2.5) Определите и реализуйте ***n***-мерную матрицу с подходящими операциями.
8. (*2.5) Реализуйте класс, подобный ***valarray***. Определите для него операции * и +. Сравните его производительность с таковой для ***valarray*** из стандартной библиотеки. Подсказка: вычисляйте выражение ***x=0.5****(***x***+***y***)+***z*** для разных размеров векторов ***x***, ***y*** и ***z***.
9. (*3) Реализуйте массив ***Fort_array*** в стиле языка Fortran, где индексы начинаются с ***1***, а не с ***нуля***.
10. (*3) Реализуйте ***Matrix***, используя для представления элементов ***vector***<***vector***<***double***> > (а не указатель на ***valarray***).
11. (*2.5) Используя композицию операций в функции (§22.4.7), реализуйте эффективную многомерную индексацию операцией []. Например, ***v1***[***x***], ***v2***[***x***][***y***], ***v2***[***x***], ***v3***[***x***][***y***][***z***], ***v3***[***x***][***y***] и ***v3***[***x***] должны порождать соответствующие элементы и подмассивы на основе простого вычисления индекса.
12. (*2) Обобщите идею из программы §22.7 в виде функции, которая получив в виде аргумента генератор случайных чисел, вывела бы графическое изображение их распределения, которое послужило бы грубой оценкой корректности генератора.
13. (*1) Если ***n*** имеет тип ***int***, каково распределение (***double***(***rand***())/***RAND_MAX***)****n***?
14. (*2.5) Выведите на экран точки внутри квадратной области. Пары координат точек должны вычисляться генератором ***Urand***(***N***), где ***N*** — число пикселов вдоль соответствующей стороны квадрата вывода. Что получающаяся картина может сказать вам о распределении случайных чисел генератором ***Urand***?
15. (*2) Реализуйте генератор ***Nrand***, дающий нормальное распределение случайных чисел.

Часть IV

Проектирование с использованием C++

Здесь представлены возможности языка C++ в более широком плане разработки программного обеспечения. Изложение фокусируется на методах проектирования и эффективной реализации проекта в терминах языковых конструкций.

Главы

23. Общий взгляд на разработку программ. Проектирование

24. Проектирование и программирование

25. Роли классов

«Я только сейчас начинаю испытывать всю сложность реализации идей на бумаге. Пока речь идет об описании чего-либо, все просто; но как только в игру вступают рассуждения, так сразу взаимосвязи, ясность и беглость изложения становятся для меня весьма нелегкой задачей, о чем ранее я и понятия не имел.»

— Чарльз Дарвин

23

Общий взгляд на разработку программ. Проектирование

Не существует серебряной пули.
— Ф. Брукс

Разработка программного продукта — цели и средства — процесс разработки — цикл разработки — этапы проектирования — выявление классов — определение операций — определение зависимостей — определение интерфейсов — реорганизация иерархии классов — модели — экспериментирование и анализ — тестирование — сопровождение программного продукта — эффективность — руководство разработкой — повторное использование — масштаб — важность личностей — гибридное проектирование — аннотированная библиография — советы.

23.1. Обзор

Данная глава является первой из трех глав, посвященных процессу разработки программ, начиная с относительно высокоуровнего взгляда на проектирование и заканчивая специфическими для языка C++ концепциями и программными технологиями, призванными непосредственно поддержать проектные решения. После введения и короткого обсуждения целей и средств программирования в §23.3, настоящая глава сосредотачивается на двух главных темах, которые можно охарактеризовать как:

- §23.4 Общий взгляд на разработку программ.
- §23.5 Практические советы по поводу организации процесса разработки программ.

В главе 24 обсуждается взаимосвязь проектирования и языка программирования. Глава 25 рассказывает о той роли, которую классы играют в организации программ на этапе проектирования. В целом эти три главы, составляющие часть IV на-

стоящей книги, призваны навести мосты между двумя крайними методиками: между высокоуровневым проектированием, мнящим себя независимым от деталей языка, и сугубо приземленным программированием, близоруко цепляющимся за мелкие детали языка. Оба конца столь широкого спектра находят свое место в рамках обширного программного проекта, но во избежание излишне высокой цены проекта или его неудачи их нужно тщательно согласовывать на предмет соответствия задач и предлагаемых методик.

23.2. Введение

Разработка и создание любого нетривиального программного продукта является весьма сложной задачей. Даже для программиста-индивидуала непосредственное написание программных конструкций является лишь частью процесса. В типичном случае, анализ исходной задачи, создание общего проекта программы, ее документация, тестирование и поддержка, а также управление всеми этими аспектами разработки намного превосходят по сложности написание и отладку отдельных фрагментов кода. Можно, конечно, назвать всю такую деятельность программированием и заявить, что «нет, я не занимаюсь проектированием, а только лишь программированием», но все равно, независимо от названий, нужно уметь и фокусироваться на отдельных частях, и уметь рассмотреть все это в целом. Ни детали, ни общая картина не должны теряться из-за стремления поскорее сдать готовую систему — хотя часто именно так и происходит.

Данная глава посвящена тем аспектам разработки программных продуктов, которые не связаны с написанием и отладкой конкретного кода для отдельных частей программы. Обсуждение указанных вопросов менее точно и менее подробно, чем рассмотрение конкретных деталей языка C++ или специфических приемов программирования, рассмотренных в нашей книге. Однако это неизбежно, ибо не существует готовых рецептов создания хороших программных продуктов. Подробные рекомендации в стиле «сделай так-то и так-то» возможны лишь для отдельных типов приложений, а не для общего случая. Ничем невозможно заменить интеллект, опыт и вкус в программировании. Поэтому данная глава предлагает лишь общие советы, альтернативные подходы и некоторые полезные предостережения.

Обсуждение затрудняется абстрактной природой программного обеспечения и тем фактом, что приемы, хорошо работающие для небольших проектов (скажем, для одного-двух человек на 10000 строк кода), не обязательно пригодны для проектов среднего или большого масштаба. По этой причине ряд положений формулируется на примере родственных тем из менее абстрактных инженерных дисциплин, а не программирования. Вам следует помнить, что «доказательства по аналогии» часто обманчивы, так что мы используем аналогии лишь для наглядной иллюстрации (а не доказательства). Переформулирование общих утверждений непосредственно в терминах языка C++ и сопутствующих примеров кода осуществляется в главах 24 и 25. Идеи же данной главы неявно подкрепляются языковыми средствами и примерами, рассмотренными во всех остальных главах книги.

Помните также, что из-за огромного разнообразия прикладных областей, людей и применяемых средств разработки не следует ожидать, что каждое высказывание непосредственно применимо к вашей конкретной задаче. Приведенные в настоя-

щей главе высказывания вытекают из опыта разработки реальных проектов и применимы к широкому кругу задач, но они не являются абсолютно универсальными. Относитесь к ним со здравой долей скептицизма.

Известно, что C++ можно использовать как улучшенный С. Однако в таком случае останутся незадействованными самые мощные средства этого языка, и лишь малая часть выгод от применения C++ будет достигнута. Данная глава фокусируется на подходе к проектированию, при котором удается эффективно применить механизм абстракции данных и средства объектно-ориентированного программирования языка C++; такой подход часто называют *объектно-ориентированным проектированием* (*object-oriented design*).

Несколько важнейших мыслей насквозь пронизывают данную главу:

- Самое важное в разработке программ — ясно понимать, что именно вы пытаетесь сделать.
- Успешный процесс разработки программного продукта — это длительный процесс.
- Разрабатываемые системы всегда эволюционируют в направлении предела сложности, который доступен нам самим и применяемым средствам разработки.
- Не существует готовых рецептов проектирования и программирования, которые могли бы заменить интеллект, опыт и вкус.
- Экспериментирование незаменимо для любого нетривиального программного проекта.
- Проектирование и программирование — итеративные процессы.
- Невозможно четко отделить друг от друга такие фазы разработки, как проектирование, программирование и тестирование.
- Не стоит рассматривать проектирование и программирование в отрыве от вопросов управления этими процессами.

Легко недооценить эти положения (это может дорого обойтись в результате). Трудно превратить эти абстрактные идеи в реальную практику. Поэтому здесь крайне важно экспериментирование. Как в самых разных областях деятельности — в судостроении, в велосипедном спорте или программировании, твердые навыки проектирования невозможно получить на основе одной лишь теории.

Часто мы забываем про человеческий фактор в проектировании и рассматриваем процесс построения программных продуктов как «последовательность строго определенных шагов, каждый из которых порождает определенные действия над входными данными, соответствующими правилам их преобразования в требуемые выходные результаты». На самом деле, любой язык программирования, любые методики проектирования и построения на основе проекта программного кода обязаны принимать во внимание человеческий фактор. Забыть про это — загубить всю работу.

Данная глава нацелена на рассмотрение вопросов проектирования систем, находящихся на пределе возможностей коллектива разработчиков (или индивидуалов). Кажется, что в самой природе людей заключается их стремление браться за задачи, находящиеся на последней грани их ресурсов. Менее амбициозные проекты слабо нуждаются в подробном рассмотрении вопросов проектирования. У них имеются

свои привычные среды и приемы разработки, от которых не стоит отказываться. Однако для совершенно новых и чрезвычайно сложных задач нужны новые, более мощные и изощренные средства и методики. А проекты, которые «мы знаем как делать», можно отдать относительным новичкам, которые еще этого не знают.

Не существует единственно верного способа проектировать и строить любые системы. Я бы даже счел веру в единственно верный способ некоторой разновидностью детской болезни, если бы не изрядное количество подверженных этой вере опытных проектировщиков и разработчиков. Пожалуйста, помните, что если некоторая методика хорошо сработала у вас в прошлом году для некоторого проекта, это не значит, что она безо всяких изменений будет столь же хорошо работать у другого разработчика или для другого проекта. Очень важно, чтобы ваш ум был всегда открыт новым обстоятельствам и идеям.

Ясно, что большая часть обсуждения относится к разработке больших систем промышленного масштаба. Читатели, не имеющие отношения к таким разработкам, могут удобно развалиться в кресле и посмотреть на «ужасные картины», которых они избежали. Или они могут сосредоточиться на подмножестве вопросов, касающихся их индивидуальных проблем. Не существует столь малых программ, для которых были бы вредны общая проработка и создание проекта работы до ее выполнения. Однако существуют столь малые программные проекты, для которых подойдут любые частные приемы проектирования, программирования и документирования. См. §23.5.2 по поводу вопросов, связанных с масштабом программ.

Самой большой проблемой в разработке программ является сложность. Для *борьбы со сложностью* есть только одно универсальное средство — *разделяй и властвуй*. Задачу, которую удается разделить на две относительно независимые подзадачи, можно считать более чем наполовину решенной. Этот простой принцип можно реализовывать разными способами. В частности, *применение модуля или класса* в проекте системы разделяет программу *две части* — на *реализацию* и ее *пользователей*, взаимодействующих (в идеале) исключительно через хорошо спроектированный *интерфейс*. Это фундаментальный подход к борьбе со сложностью программ. Аналогично, процесс проектирования программного продукта может быть разбит на отдельные действия с четко определенными взаимодействиями между участвующими в проекте людьми. Это фундаментальный подход к борьбе со сложностью процесса разработки и связанными с ним человеческими отношениями.

Для обоих случаев *вычленение отдельных частей и определение интерфейса их взаимодействия* требует максимального опыта и тонкого вкуса. Такое вычленение — это вовсе не простой механический процесс, а нечто, требующее проникновения в самую суть проблемы, для чего требуется глубокое понимание системы на подходящем уровне ее абстракции (см. §23.4.2, §24.3.1 и §25.3). Близорукий взгляд на процесс разработки программ часто приводит к их серьезным изъянам. Отметим также, что в *реальной жизни любое разделение*, будь-то частей программы или людей, *выполняется значительно легче, чем их эффективное взаимодействие* по разные стороны барьера без разрушения этого барьера и с предоставлением средств коммуникации, достаточных для эффективной кооперации.

Настоящая глава рассматривает именно подход к проектированию, а не метод проектирования целиком, что очевидным образом выходит за рамки данной книги. Представленный нами подход может использоваться с разной степенью формализации, и он может служить основой для разных степеней формализации. Аналогич-

но, данная глава не служит литературным обзором по проектированию и разработке программных продуктов, так что в ней не затрагиваются все относящиеся к изучаемой проблеме вопросы и не перечисляются все известные точки зрения. Такой обзор представлен в книге [Booch, 1994]. Используемые в данной главе термины являются общепринятыми и широко употребляемыми. Особо повторяемые термины, такие как *проект* (*design*), *прототип* (*prototype*) и *программист* (*programmer*) имеют в литературе множество разных и зачастую противоречащих друг другу определений. Пожалуйста, постарайтесь не попасться на удочку узко понимаемых определений и не прочтите в данной главе чего-нибудь такого, чего я вовсе не намеревался сказать.

23.3. Цели и средства

Целью профессионального программирования является изготовление и поставка законченного программного продукта, который удовлетворит заказчиков и/или иных его пользователей. Главным средством достижения такой цели является четкое понимание внутренней структуры программы, для чего требуется вырастить группу проектировщиков и программистов, достаточно квалифицированную и хорошо мотивированную для быстрого реагирования на новые обстоятельства и возможности.

Почему так? Ведь вроде бы внутренняя структура программы и детали процесса ее изготовления не должны волновать заказчика (пользователя). Более того, если конечный пользователь вынужден беспокоиться о таких деталях, то с программой явно что-то не в порядке. Тогда почему же внутренняя структура программа и качество изготавливающего программу коллектива разработчиков так важны?

Ответ на этот вопрос таков — ясная внутренняя структура программы нужна для того, чтобы ее было легче (быстрее и надежнее):

- тестировать,
- портировать (переносить на другие системы),
- сопровождать и поддерживать,
- расширять,
- модифицировать,
- понимать.

Главная мысль состоит в том, что каждая успешная программа имеет определенное время жизни, в течение которого с ней работают разные группы проектировщиков и программистов, она переносится на новые виды аппаратуры, подстраивается под непредвиденные обстоятельства и, вообще, так или иначе периодически изменяется. За время жизни программного продукта своевременно появляются его новые версии с приемлемым уровнем ошибок. Не планировать указанных аспектов поведения программного продукта — заранее обрекать его на неудачу.

Далее, хотя конечные пользователи и не обязаны знать внутреннюю структуру программной системы, они вполне могут сами захотеть это сделать. Например, пользователь может захотеть узнать поподробнее внутренний дизайн системы для того, чтобы оценить ее надежность и/или способность к модификациям и расширению. Если программный продукт не представляет собой завершенной системы, например,

это может быть набор вспомогательных библиотек, то в этом случае пользователи захотят узнать побольше деталей с целью более эффективного их использования.

В реальных разработках приходится искать разумный баланс между отсутствием общего проекта (четкой внутренней структуры) и слишком большим упором на архитектурные аспекты. Первая крайность приводит к бесконечному «мы только сдадим эту программу, а все остальные проблемы решим в следующей версии». Вторая же крайность приводит к переусложненным проектам, в которых суть скрывается за формализмом и часто задерживается выпуск программ («новая структура программы *намного* лучше старой; люди с удовольствием подождут»). Кроме того, это часто приводит к излишним требованиям максимальных компьютерных ресурсов, которые пользователи не всегда могут себе позволить. *Такое балансирование — самый сложный аспект проектирования*, в котором ярче всего раскрываются настоящие таланты и настоятельно требуется изрядный опыт. Этот выбор труден даже для индивидуального проектировщика и программиста, и его тем более сложно сделать в больших программных проектах, в которых задействовано множество людей с разным опытом и квалификацией.

Программы должны производиться и обслуживаться организациями, способными делать это несмотря на смену персонала и менеджеров. Часто для этого стараются осуществлять разработку так, что вся работа разбивается на узкоспециализированные слои, жестко встраиваемые в общую структуру разработки. Для этого формируется множество быстро обучаемых (дешевых) и взаимозаменяемых программистов низкого уровня (кодировщиков — «coders») и множество не столь дешевых, но столь же взаимозаменяемых проектировщиков. От кодировщиков не требуется принимать архитектурных решений, в то время как проектировщики не обязаны разбираться в мелких и низкоуровневых деталях кода. Такой подход, однако, нередко ведет к неудачам. Там, где он все-таки работает, получаются переусложненные системы с плохой производительностью.

Проблемы этого подхода заключаются в следующем:

- *недостаточное взаимодействие и взаимопонимание* между проектировщиками и кодировщиками приводит к *упущенным возможностям*, задержкам, неэффективности и повторению проблем из-за того, что никто не накапливает предыдущий опыт и не учится на ошибках;
- *узкие рамки* работы специалистов *сковывают инициативу*, мешают профессиональному росту, способствуют разболтанности и высокой текучке кадров.

По сути дела, *такой системе не хватает каналов обратной связи*, чтобы люди могли плодотворно общаться и учиться на опыте других. Это растрата человеческого таланта. Создание среды и структуры осуществления процесса разработки программ, в которых люди проявляют таланты, развивают способности, предлагают свои идеи и радуются результатам — это не только благородная цель, достойная сама по себе, но и практическая вещь, приносящая экономическую выгоду.

С другой стороны, строить программную систему, документировать и поддерживать ее в безусловном порядке невозможно без некоторой бюрократической организационной структуры. Просто найти подходящих людей и дать им наброситься на задачу наилучшим с их точки зрения образом часто бывает идеальным началом для инновационных проектов. Дальше, по мере развития проекта, растет необходимость в планировании, специализации и установлении более формальных связей

между участниками проекта. Под словом «формальных» я не имел в виду нечто строго математическое (хотя в отдельных случаях и это не исключается), а скорее набор рекомендаций по именованию, документированию, тестированию и т.д. Тут, естественно, требуется вкус и чувство меры. Слишком жесткая организация может придушить рост и творчество. Здесь как раз и испытываются талант и опыт руководителя. Для индивидуальных программистов дилемма заключается в том, чтобы выбрать между изобретательством и копированием известных «книжных образцов».

Моя рекомендация — думать не только о ближайшем выпуске, но и о более долгосрочных проблемах. Беспокоиться исключительно о ближайшем выпуске программы — значит планировать неудачу. Нужно разрабатывать стратегию организации процесса разработки программных продуктов, нацеленную на множество выпусков множества продуктов, то есть нужно планировать целую серию успехов.

Цель проектирования состоит в построении ясной и относительно простой *внутренней структуры программы*, иногда называемой *архитектурой*. Другими словами, мы создаем инфраструктурную среду, в которую должны вписываться отдельные части программного кода, и которая сама направляла бы написание отдельных фрагментов.

Проект — это конечный результат проектирования (если, конечно, у итеративного процесса есть конечный результат). Именно проект организует взаимодействие между проектировщиком и программистом, а также между разными группами программистов. Тут возможны две крайности. Для индивидуального программиста и небольшой задачки, которую нужно решить за пару дней, проект может заключаться в нескольких набросках на обратной стороне почтового конверта. В то же время, для разработок, в которых участвуют сотни проектировщиков и программистов, проект может вылиться во многие тома спецификаций, тщательно выполненных в соответствии с устоявшимися (формальными или полуформальными) стандартами. Выявление подходящего уровня детализации, точности и формализации проекта — сама по себе непростая техническая и управленческая задача.

В данной и двух последующих главах книги я предполагаю, что проект системы заключается в наборе объявлений классов (в типичном случае с опущенными закрытыми секциями объявлений как несущественными подробностями) и указании связей между ними. Это, конечно, упрощение. Для реального проекта имеют значение огромное множество деталей, таких как возможность параллельного исполнения кода, группировка идентификаторов в пространства имен, применение глобальных функций и данных, параметризация классов и функций, реорганизация структуры программы с целью минимизации перекомпиляций, возможность работы в распределенной сетевой среде и т.д. Тем не менее, наше упрощение оправдано, так как позволяет сосредоточиться на том уровне проектной детализации, для которого ключевое значение имеют классы языка C++. Остальные проектные особенности частично затрагиваются в данной главе, а те из них, что напрямую связаны с особенностями языка C++ — в главах 24 и 25. За более детальными сведениями об объектно-ориентированном проектировании обратитесь к книге [Booch, 1994].

Я не сосредотачиваюсь на различиях между анализом и проектированием, поскольку эта тема выходит за рамки настоящей книги, и она, к тому же, очень чувствительна по отношению к разным проектным методикам. Важно так выбрать метод анализа, чтобы он соответствовал методу проектирования, а метод проектирования — чтобы соответствовал стилю программирования и языку программирования.

23.4. Процесс разработки

Разработка программного продукта — процесс итеративный. За время разработки многократно возвращаются к одним и тем же промежуточным стадиям, каждый раз усовершенствуя соответствующие им состояния проекта. Вообще говоря, такой процесс не имеет конца. Начиная работать, вы обычно обращаетесь к некоторой базе ранее выполненных другими людьми проектов, к библиотекам и прикладным средам разработки. Заканчивая проект, вы оставляете его другим для улучшений и расширений, а также для переноса на иные платформы. Естественно, конкретный проект должен иметь определенные начало и конец, и желательно (хотя это и трудно) четко планировать процесс разработки программного продукта во времени. Думая, что стартуете с чистого листа, вы лишь затрудняете себе работу. Полагая, что все в мире заканчивается с последним выпуском вашей программы, вы ставите преемников в затруднительное положение (и себя самого в этом качестве).

Последующие разделы можно читать почти что в произвольном порядке, так как аспекты проектирования и реализации могут произвольно перемежаться в реальных проектах. То есть проект почти всегда переделывается, отталкиваясь от опыта предыдущих проектов и их реализаций. Кроме того, его ограничивают календарные планы, квалификация персонала, необходимость обеспечить совместимость с предыдущими версиями или иными программами и т.д. Очень важно для проектировщиков, менеджеров и программистов внести, тем не менее, в процесс разработки определенный порядок, не удушающий творчество и не разрушающий обратные связи между людьми, способствующие конечному успеху дела.

Процесс разработки можно разделить на три этапа:

- *анализ*: выявление границ решаемой задачи (проблемы);
- *проектирование*: создание общей внутренней структуры системы;
- *реализация*: написание кода и тестирование программы.

Пожалуйста, не забывайте об итеративной природе процесса — важно, что перечисленные этапы не пронумерованы. Также стоит отметить, что некоторые аспекты разработки программ не подпадают под разбиение на этапы:

- Экспериментирование
- Тестирование
- Анализ проекта и реализации
- Документирование
- Управление проектом (менеджмент)

Поддержка и сопровождение программы сводятся к дополнительным проходам по всему циклу разработки (§23.4.6).

Очень важно, чтобы анализ, проектирование и реализация не слишком отдалялись друг от друга, и чтобы люди, ответственные за разные виды деятельности, могли при этом взаимодействовать и обмениваться мыслями и опытом. Слишком часто в больших проектах это отсутствует. В идеале, в процессе выполнения проекта люди должны изменять свое амплуа и переходить от одного этапа к другому — только так можно осуществить перенос важной и тонкой информации, хранящейся в голове человека. К сожалению, многие организации выставляют барьеры на пути

таких человеческих переходов, предоставляя, например, проектировщикам более высокий статус и/или зарплату, чем «простым программистам». Если уж среди сотрудников не практикуются переходы туда-сюда с целью учить и учиться, то желательно поощрять их беседовать с участниками «других» этапов разработки.

Для небольших и средних проектов часто не делается различия между анализом и проектированием; эти две фазы сливаются в одну. А для маленьких проектов не делается различия и между проектированием и реализацией. При этом, конечно, автоматически решаются «проблемы общения». Любому проекту важно придать подходящий уровень формализации и поддерживать соответствующую степень разделения этапов (§23.5.2). Тут не существует единственно верного решения.

Описанная здесь модель разработки программных продуктов радикально отличается от традиционной «модели водопада». В последней процесс разработки разворачивается в линейную последовательность шагов от анализа до тестирования. «Модель водопада» имеет серьезный изъян, заключающийся в том, что информация в рамках этой модели «течет» лишь в одном направлении. Когда проблемы обнаруживаются «внизу по течению», возникает сильное методологическое и организационное давление с тем, чтобы решать их на месте выявления, не затрагивая предыдущие этапы процесса. Такое намеренное игнорирование обратных связей обедняет проектное решение, а исправления «по месту возникновения» часто порождают ущербные программы. В случаях, когда информация с неизбежностью вытекает назад, приходится изменять проект, что замедляет разработку и вызывает нескладный волнообразный эффект в системе, предполагающей отсутствие таких изменений, а потому сопротивляющейся и медленно реагирующей на изменения. В качестве довода в пользу отсутствия изменений и решения проблем на месте приводят суждение о том, что одно подразделение не должно ради своего удобства перекладывать работу на плечи другого подразделения. Наконец, когда выявляется проблема, исключительно плохо преодолеваемая на месте, оказывается, что к этому моменту выполнена грандиозная бумажная работа по документированию, переделать которую столь сложно и накладно, что лучше уж любой ценой и как-нибудь все исправить по-прежнему «на месте». Таким образом, бумажная работа оказывается важнейшим фактором в разработке программного продукта. Разумеется, при любой организации работ такие проблемы могут и наверняка возникают в рамках больших программных проектов. Ясно, что какие-то бумаги действительно необходимы. Однако применение для разработки линейной модели (модели «водопада») увеличивает вероятность того, что бумажные проблемы могут выйти из под контроля.

Недостаток модели «водопада» заключается в недостаточности обратной связи и неспособности реагировать на изменения. Опасность же описанного ранее итеративного подхода заключается в возможной попытке подменить необходимость думать серией плохо сходящихся изменений. Обе проблемы легче описать, чем решить, и в любом случае имеется соблазн счесть бурную деятельность гарантией успешного продвижения к желаемой цели. Естественно, что по мере продвижения проекта разные его фазы становятся более или менее важными. В самом начале акцент делается на анализе и проектировании, а вопросы написания конкретного кода в этот момент не столь актуальны. Постепенно акцент перемещается на связку проектирование-программирование, а затем и на связку программирование-тестирование. Однако никогда нельзя 100%-но фокусироваться на отдельных стадиях, забывая про все остальные.

Помните, что *если вы не знаете, чего хотите достичь, ничего не поможет* — ни внимание к деталям, ни правильные приемы менеджмента, ни передовые методики проектирования, ничего. Большая часть проектов проваливается как раз по причине отсутствия четко определенной и реалистичной цели. Нужно ясно понимать свою цель, четко определять промежуточные фазы разработки и не пытаться подменять техническими приемами отсутствие перечисленного. При наличии четкой цели используйте все доступные методы, даже если они требуют серьезных инвестиций; люди работают лучше в условиях хорошего финансирования и благоприятной социальной и технологической среды. Однако не думайте, что следовать этому совету легко.

23.4.1. Цикл разработки

Разработка программной системы является итеративным процессом. Основной цикл заключается в повторе следующих шагов:

0. Исследование проблемы.
1. Создание общего проекта.
2. Нахождение стандартных компонентов. Приспособление компонентов к проекту.
3. Создание новых стандартных компонентов. Их приспособление к проекту.
4. Сборка проекта.

Для аналогии рассмотрим автомобильный завод. Чтобы запустить производство, нужен полный общий проект нового типа автомобиля. Самый первый набросок будет отталкиваться от предварительного анализа и состоять в словесном описании, относящемся скорее к области применения автомобиля, а не к техническим деталям его устройства, позволяющим добиться заявленных целей. Принятие решения о требуемых свойствах автомобиля является самым трудным решением. *Правильно принятое решение* часто является *плодом озарения одной сверхпроницательной личности*. Многие проекты испытывают дефицит четко заявленных целей, и по этой причине они часто терпят неудачу.

Допустим, что мы хотим построить среднего размера седан с четырьмя дверями и довольно мощным двигателем. После этого совершенно необязательно набрасываться на выполнение проекта автомобиля (и его узлов) с нуля. Аналогично, программисту на этом этапе также нет смысла сразу же составлять программный проект и начинать писать код.

Намного лучше сначала посмотреть, какие компоненты уже имеются на заводе или у надежных поставщиков. Найденные таким образом компоненты не обязаны в точности подходить для нового автомобиля. Их всегда можно будет в дальнейшим приспособить для этой цели. Неплохо сразу же написать спецификации для будущих версий этих компонентов, чтобы они лучше подходили для нового автомобиля. Например, пусть имеется двигатель со всеми подходящими характеристиками, кроме слегка недостаточной мощности, но есть возможность слегка турбировать этот двигатель для достижения желаемой мощности без внесения изменений в базовую конструкцию. Выполнить такое турбирование без внесения изменений в базовый проект двигателя можно лишь тогда, когда такой аспект модификаций был учтен при выполнении исходного проекта двигателя. Ясно, что эта форма доработки двигателя выполняется во взаимодействии с поставщиком двигателя.

Проектировщик программного продукта и программист находятся в схожем положении. Например, они могут *использовать полиморфные классы и шаблоны для подгонки готовых компонентов.* Но, конечно, не следует думать, что можно осуществить произвольную подгонку, если только разработчик компонента *заранее не заложил в него эту возможность.*

Выявив наличие и определив возможности готовых стандартных компонентов, разработчик не бросается на проектирование новых (недостающих) компонентов для автомобиля. Например, оказалось, что нигде нет подходящего кондиционера, а под капотом есть свободное пространство L-образной формы. Прямолинейным решением будет разработка кондиционера L-образной формы. Но вероятность приспособить в дальнейшем такую экзотику под другие машины (даже после сильной подгонки) крайне мала. В результате не удастся снизить совокупную стоимость разработки за счет тиражирования решения на большую партию самых разных машин, а эффективная жизнь нашего кондиционера будет весьма короткой. В результате более мудрым решением будет разработка собственного кондиционера традиционных форм и габаритов с широкими возможностями приспособления к разным конкретным случаям и нуждам. Это, конечно, займет больше времени и сил, чем проработка частного L-решения, и возможно повлечет за собой внесение изменений в первоначальный дизайн разрабатываемого нового автомобиля, чтобы он мог принять кондиционер более традиционной формы и габаритов. Возможно даже, что более традиционный кондиционер после этого снова нужно будет довести (в соответствии с заложенными в его проект вариантами модификации), чтобы сделать его великолепно подходящим новому автомобилю. И снова отмечаем, что проектировщик системы и программист находятся в аналогичном положении, и им тоже не следует сразу же писать узкоспецифичный код вместо разработки универсального решения, которое имеет шанс стать общеупотребительным стандартом в будущем.

Наконец, исчерпав потенциал готовых стандартных компонентов, мы собираем «окончательный» проект. Мы используем в нем столь малое количество специфических компонентов, насколько это возможно, ибо в уме мы держим простую мысль, что уже в следующем году нам придется все начинать сначала, разрабатывая очередную модель автомобиля, и специфические компоненты при этом придется либо сильно переделывать, либо выбросить совсем. Печально, но опыт разработки программных компонентов говорит о том, что немногие части программы удается хотя бы выделить в четко очерченные компоненты, и еще меньше из них находит свое применение за пределами оригинального проекта.

Я не утверждаю, что все проектировщики автомобилей поступают столь рационально, как мы это описали, и не все проектировщики программ допускают перечисленные выше ошибки. Я, просто, твердо полагаю, что описанную (автомобильную) модель можно и нужно с разумными поправками использовать в процессе разработки программных продуктов. Данная глава и последующая рассматривают технологии, позволяющие применять описанную модель в рамках языка C++. Но я все же подчеркиваю, что из-за нематериальной природы компьютерных программ трудно полностью избежать указанных ошибок (§24.3.1, §24.3.4), и в §23.5.3 я показываю, что от использования приведенной здесь модели людей часто удерживают корпоративные стандарты.

Отметим, что эта модель разработки хороша именно в долгосрочной перспективе. Если ваш горизонт распространяется только до ближайшего выпуска, разработ-

ка и создание стандартных компонентов теряет для вас всякий смысл. Это будет выглядеть для вас излишними проблемами (и затратами). Наша модель предлагается для организаций, трудовая жизнь которых распространяется на многие выпуски крупных программных продуктов, чтобы могли оправдать себя дополнительные инвестиции в инструменты (менеджмента, проектирования и программирования) и обучение персонала (менеджеров, проектировщиков и программистов). Можно считать это наброском фабрики по производству программных продуктов. Забавно, но он отличается лишь масштабом от реальной практики талантливых и предусмотрительных индивидуальных программистов, которые с годами набирают отличный запас приемов, проектов, инструментов и библиотек, резко усиливающих их персональную производительность. Похоже, что многим организациям не удается воплотить в жизнь работающие методики индивидуальных программистов из-за близорукости или неумения управлять ими в более крупных масштабах.

Отметим, что совершенно *неразумно ожидать* от стандартных компонентов *абсолютной универсальности*. Могут появиться несколько международных стандартных библиотек, но большинство из них останется стандартными лишь в рамках конкретных стран, отраслей, организаций, отделов и прикладных областей. Мир просто слишком велик, чтобы ограничиваться немногими универсальными стандартами.

Стремление к чрезмерной универсальности на начальных стадиях самых первых проектов почти наверняка повлечет за собой то, что эти проекты никогда не будут завершены. Цикл разработки потому и цикл, что нужно черпать опыт из полученных в рамках очередной итерации работающих систем (§23.4.3.6).

23.4.2. Цели проектирования

Каковы задачи, стоящие перед проектированием? Безусловно достижение простоты, но простоты в каком смысле, ведь проект должен развиваться? Он будет расширяться, портироваться, настраиваться и вообще меняться столь большим числом способов, что их невозможно предвидеть. Следовательно, мы должны стремиться к тому, чтобы в проект системы и ее реализацию было легко вносить изменения. Нет ничего удивительно в том, что требования к системе и изменения в проект будут несколько раз вноситься даже при разработке самого первого выпуска продукта.

Из всего этого следует, что система должна проектироваться таким образом, чтобы оставаться сколь возможно простой под воздействием серии модификаций. Мы должны закладывать изменения в проект, то есть мы должны стремиться к:

- гибкости,
- расширяемости,
- переносимости.

Лучше всего это достигается попыткой изоляции и *инкапсулирования тех частей системы*, которые *более всего подвержены потенциальным изменениям*, с тем чтобы в дальнейшем проектировщики и программисты вносили изменения в четко очерченные отдельные части системы. Это выполняется путем выявления ключевых концепций разработки и соотнесения их с определенным классом, в исключительное ведение которого передается вся работа с информацией, относящейся к концепции. В таком случае изменения можно сосредоточить в единственном классе.

В идеале, необходимые изменения в концепции можно было бы выполнять с помощью введения производного класса (§23.4.3.5) или передачей нового параметра шаблону. Конечно, *идеал легче провозгласить, чем реально осуществить.*

Рассмотрим пример. В программе моделирования погодных явлений возникла необходимость визуализировать тучу. Как нам сделать это? Мы не можем написать внешнюю процедуру для показа тучи, поскольку ее вид зависит от ее внутреннего состояния, а о нем должна знать только сама туча.

Первое решение — пусть туча сама себя показывает. Во многих ограниченных случаях такой стиль решения проблемы является приемлемым. Он, однако, не является универсальным, ведь можно по-разному показывать тучу; детальным образом, схематично, в виде иконки на карте и т.д. Другими словами, вид тучи зависит не только от самой тучи, но и от окружающего контекста.

Второе решение — пусть туча знает окружающий контекст и сама себя показывает. Это решение более универсально. Но оно по-прежнему не является общим решением. Оно нарушает принцип, заключающийся в том, что туча знает все о себе и ни о ком другом, а за каждую отдельную концепцию в программе отвечает какой-то определенный класс. Также вряд ли возможно согласованное определение понятия «окружение тучи», поскольку внешний вид тучи зависит даже от наблюдателя. Даже в обычной жизни то, как я вижу тучу, зависит от того, смотрю ли я на нее невооруженным глазом, или через поляризационный фильтр. Также этот вид зависит и от определенного общего фона, например от положения солнца. Добавление иных объектов, таких как другие облака или самолеты, еще более усложняет проблему. А чтобы еще более усложнить задачу проектировщику, добавьте возможность присутствия нескольких наблюдателей.

Третье решение — пусть туча и все остальные объекты, такие как солнце или самолеты, докладывают о себе наблюдателю. Это решение еще более универсально. Но оно может оказаться слишком сложным, в том числе с точки зрения необходимости огромных вычислительных мощностей и иных компьютерных ресурсов. Например, как понятным образом организовать данные, передаваемые наблюдателю перечисленными выше объектами?

Тучи не являются частыми гостями программных проектов (для примера см. §15.2), но вот другие объекты, которые вовлекаются во множество операций ввода/вывода, встречаются часто. Это делает пример с тучей вполне уместным в контексте разработки программ и особенно в проектировании библиотек. Код C++ для логически похожего примера встречается в контексте манипуляторов, используемых для форматированного вывода в библиотеке потоков (§21.4.6, §21.4.6.3). Заметьте, что третье решение не является самым «правильным» решением — оно лишь самое универсальное. Проектировщик должен балансировать между множеством различных требований, чтобы выбрать оптимальный уровень общности/абстрактности, подходящий для конкретной проблемы в рамках данной системы. Выскажем грубое эмпирическое соображение, что оптимальным уровнем абстракции для программ длинного цикла жизни является такой, который предоставляет наибольшую универсальность, которую вы только можете себе позволить и воспринять. Чрезмерная универсальность, выходящая за рамки возможностей индивидуального понимания и технических рамок проекта, причинит один лишь вред: вызовет задержки, приведет к неприемлемо низкой производительности, неуправляемому проекту и, в конце концов, просто к катастрофе.

Чтобы сделать приемы проектирования экономически приемлемыми и управляемыми, мы должны учитывать необходимость повторного использования проекта (§23.5.1) и не забывать совсем уж об эффективности (§23.4.7).

23.4.3. Этапы проектирования

Рассмотрим проектирование отдельного класса. В общем случае, это не слишком хорошая идея. *Концепции не существуют в полной изоляции*; скорее, концепции определяются в контексте других концепций. Точно так же, и *классы не существуют изолированно*. Они определяются совместно с логически связанными с ними классами. Как правило, мы работаем с набором логически связанных классов. Такой набор классов принято называть *библиотекой классов* или *компонентом* (*class library* или *component*). Иногда все классы компонента составляют единую иерархию классов, иногда — определяются в единственном пространстве имен, а иногда они являются просто произвольным набором объявлений (§24.4).

Набор классов объединяется в компонент согласно некоторому логическому критерию, часто по общему стилю, но еще чаще по общему сервису. Таким образом, *компонент* — это *единица проекта, документации и объект повторного использования*. Это не значит, что если применяете один класс из компонента, вы должны понимать код всех остальных классов и использовать их, загружая вхолостую память машины. Наоборот, мы стремимся использовать классы с минимальными затратами машинных ресурсов и человеческих усилий. Однако чтобы уверенно пользоваться каким-либо классом компонента, нам нужно понять объединяющий классы логический критерий (хорошо, если он ясно задокументирован), соглашения и стиль программирования, а также использование общих ресурсов (если это имеет место).

Итак, рассмотрим, как можно было бы спроектировать компонент. Поскольку часто это непростая задача, разобьем ее на отдельные шаги, чтобы сфокусироваться на отдельных подзадачах логически полным образом. Как обычно, не существует единственного способа сделать такое разбиение. Вот, однако, последовательность шагов, которая помогла многим:

1. Выявите концепции/классы и их фундаментальные взаимосвязи.
2. Уточните классы, определив набор их операций.
   - Классифицируйте эти операции, в том числе определите необходимость конструкторов, деструкторов и операций копирования.
   - Обеспечьте минимальное решение, достаточно полное и удобное.
3. Уточните классы в плане их зависимостей.
   - Параметризация, наследование и иные типы зависимостей.
4. Определите интерфейсы.
   - Разделите функции на открытые и защищенные.
   - Определите точный тип операций над классом.

Помните, что это шаги итеративного процесса. Как правило, чтобы получить удобный в работе проект, нужно неоднократно пройтись по этим шагам. Одним из преимуществ качественно выполненного анализа и абстракции данных является относительная простота изменения взаимозависимостей классов даже после того,

как написан конкретный код программы. Это, конечно, не совсем тривиальная задача.

После этого мы реализуем код классовых методов и пересматриваем проект, учитывая опыт, полученный при реализации кода. В последующих подразделах данной главы мы обсудим все эти шаги последовательно, один за другим.

23.4.3.1. Этап 1: отражение концепций классами

Выявите концепции/классы и их фундаментальные взаимосвязи. Ключ к хорошему проекту — это непосредственное *моделирование реальностей прикладной задачи в виде классов*, представление их взаимосвязей известными способами, например в виде наследования, причем все это нужно периодически повторять на разных уровнях абстракции. Но как на практике нужно выискивать концепции/классы? Как понять, какие именно классы нам нужны?

Лучше всего начать такой поиск прямо в самом приложении, а не рыться в портфеле абстракций и концепций компьютерного ученого. Послушайте того, кто собирается стать профессиональным пользователем вашей системы, или того, кто недоволен существующей системой. Обратите внимание на термины, которыми они при этом оперируют.

Часто говорят, что существительные соответствуют классам и объектам программы (и это часто соответствует действительности). Ясно, однако, что это далеко не конец истории. Глаголы соответствуют операциям над объектами, традиционным (глобальным) функциям, возвращающим новые значения на базе их значений-аргументов, и даже классам. Последний случай иллюстрируется классами функциональных объектов (§18.4) и манипуляторами (§21.4.6). Такие глаголы, как «итерировать» и «выполнить (как единое целое)» соответствуют поведению итераторных объектов и объектов, отвечающих за целостное исполнение группы операций (транзакций) над базами данных. Иногда даже прилагательные можно с пользой для дела представлять классами. Например, прилагательные «хранимый», «параллельный», «зарегистрированный» или «связанный» могут быть представлены проектировщиком в виде классов, которые в дальнейшем в качестве виртуальных базовых классов (§15.2.4) помогут придать классам иерархии желательные дополнительные атрибуты.

Не всякий класс имеет прямое соответствие реальностям прикладной задачи. Некоторые представляют системные ресурсы и абстрактные детали реализации (§24.3.1). Также важно отойти от близкого соответствия моделям старой программной системы. Например, вряд ли в новой системе нужно подражать старой системе, которая для работы с данными помогала пользователям моделировать «перекладывание бумажек».

Наследование применяется для *отражения общности концепций.* Что самое важное, наследование должно отражать иерархические отношения классов, моделирующих индивидуальные концепции (§1.7, §12.2.6, §24.3.2). Иногда это называют *классификацией* (*classification*) или *таксономией* (*taxonomy*). Активно выискивайте общности. Обобщение и классификация — это абстрактная деятельность, требующая проникновения в самую суть вещей для получения полезных и надежных результатов. Общий базовый класс должен отражать самую общую концепцию, а не просто похожую концепцию с минимальными данными для представления.

Отметим, что классификации должны быть подвергнуты те аспекты концепций, которые мы моделируем в нашей системе, а не аспекты, имеющие значение в других случаях. Например, в математике окружность — это частный случай эллипса, но в большинстве программ окружность не моделируют в свете ее связи с эллипсом, и эллипс не представляют в виде типа, производного от типа окружности. Тут такие аргументы, что мол так обстоит дело в математике, не работают. Для большинства программ ключевое свойство окружности — это равноудаленность ее точек от центра окружности. Это свойство должно поддерживаться всеми операциями типа окружности (то есть это свойство является инвариантом; §24.3.7.1). С другой стороны, эллипс характеризуется наличием двух фокальных точек, которые во многих программах можно изменять независимым образом. Если эти точки совмещаются, эллипс становится похож на окружность, но это не окружность, поскольку его операции не сохраняют инварианта окружности. В большинстве систем это отличие будет отражено в разных наборах операций для окружности и эллипса, так что один набор не является подмножеством другого.

Мы не просто определяем набор классов и отношений между ними и используем их в окончательной системе — мы на самом деле создаем первоначальный набор классов и их взаимоотношений с тем, чтобы многократно совершенствовать этот набор (§23.4.3.5) и в итоге получить универсальный, гибкий и стабильный набор, помогающий на всех будущих стадиях эволюции системы.

Наилучшим инструментом для поиска и выявления концепций/классов является обычная классная доска. Уточнение начального набора лучше всего выполнять в процессе дискуссий со специалистами в предметной области задачи и некоторыми друзьями. Дискуссии нужны для выявления *первоначального словаря задачи* и *базовой системы концепций*. Редкие люди могут делать это в одиночку. Дальше с целью уточнения и улучшения классов из первоначального набора стоит попытаться смоделировать систему, используя проектировщиков в роли классов. Это может выявить абсурдность некоторых первоначальных идей, стимулировать предложение альтернатив и создать согласованное понимание системы у всех участников. Для документирования дискуссии можно применить, например, CRC-карточки (Class, Responsibility, Collaborators) [Wirfs-Brock, 1990], которые так называются в соответствии с записываемой на них информацией (Класс, Ответственность, Сотрудники).

Примеры использования (*use cases*) — это описание частных случаев использования системы. Вот простой пример использования телефонной системы: поднимаем трубку, набираем номер, телефон на другом конце звонит, там снимают трубку. Разработка набора примеров (частных случаев) использования чрезвычайно полезна на всех этапах разработки. На начальном этапе выявление полезных частных случаев применения нашей системы помогает нам глубже понять, что именно мы хотим разработать. На этапе проектирования они помогают нам интроспектировать систему (например, через CRC-карточки) с целью убедиться, что относительно статическое описание системы в терминах классов и объектов имеет смысл с точки зрения пользователя системы. На этапах программирования и тестирования примеры использования становятся источниками тестов. Таким образом, примеры частных случаев использования вносят ортогональный взгляд на систему с точки зрения ее соответствия реальности.

Примеры использования описывают систему как (динамическую) работающую сущность. Поэтому они могут сконцентрировать внимание проектировщика на

функциональных аспектах системы и отвлечь его внимание от поиска важных концепций, отображаемых в классы. Это особенно опасно в том случае, когда проектировщик имеет большой опыт в структурном анализе и недостаточный опыт в объектно-ориентированном проектировании/программировании, так что упор на частных примерах использования может привести в итоге к функциональной декомпозиции. Вообще, набор примеров использования — это не проект. *Примеры использования* системы должны быть *уравновешены анализом структуры системы.*

Команда разработчиков также может попасться в ловушку погони за полным набором примеров использования. Такие ошибки дорого обходятся. Как и в случае поиска важных сущностей и соответствующих им классов приходит время остановиться и сказать: «Хватит! Наступил момент испробовать то, что мы уже получили и посмотреть, как это работает». Применяя подходящий набор классов и подходящий набор примеров использования можно двигаться вперед, приобретая опыт и обратную связь для корректировки проекта. Всегда трудно решить, когда же нужно остановиться. Особенно трудно остановиться тогда, когда известно, что позже все равно придется вернуться для завершения проекта.

Сколько нужно примеров? В общем случае на этот вопрос ответить невозможно. Тем не менее, в конкретном проекте всегда приходит время, когда становится ясно, что большая часть стандартной функциональности системы уже отражена в текущем варианте проекта, а некоторая часть специфического поведения и обработки ошибок до некоторой степени изучена. Здесь и наступает момент, когда нужно переходить к следующим этапам проектирования и программирования.

Когда вы пытаетесь оценить процент покрытия системы набором примеров использования, полезно разделить все имеющиеся примеры на первичные и вторичные. Первичные описывают наиболее общие и нормальные действия, а вторичные — менее обычные действия и сценарии обработки ошибок. Часто говорят, что когда 80% первичных примеров и некоторое количество вторичных описаны, можно заканчивать сбор примеров, но поскольку мы не знаем, что такое 100%, то эту рекомендацию следует воспринимать как ориентировочную. Опыт и вкус сослужат здесь неоценимую пользу.

Концепции, операции и взаимосвязи естественным образом вытекают из нашего понимания прикладной области задачи или появляются в процессе практической работы с существующей классовой структурой. Они составляют основу фундаментального понимания приложения. Часто они порождаются классификацией, такой как, например, «машина с раздвижной лестницей» есть частный случай пожарной машины, которая есть частный случай грузовика, который есть частный случай колесного транспортного средства. Разделы §23.4.3.2 и §23.4.5 знакомят с несколькими способами взгляда на классы и классовые иерархии с точки зрения их улучшения.

Остерегайтесь излишней графической инженерии. В некоторый момент вас попросят представить проект кому-нибудь в графической форме, раскрывающей структуру разрабатываемой системы. Это может оказаться для вас положительным опытом, так как заставит сосредоточить внимание на том, что действительно важно для системы и заставит представить все это в форме, понятной окружающим. Презентации можно рассматривать как ценное проектное средство. Подготовка презентации для людей, имеющих к проекту интерес и возможность порождать конструктивные замечания, является испытанием для качества разработанных вами концепций и способности ясного изложения идей.

В то же время, формальная презентация проекта также имеет и отрицательные стороны, ибо подталкивает к тому, чтобы представить проект идеальной системы, а не той, которая у вас уже есть или которую можно произвести в реальные сроки. Когда конкурируют разные подходы, а начальство на самом деле не понимает деталей или не заботится о них, презентация превращается в ложное соревнование, в котором побеждает та команда, которая представит наиболее грандиозный проект ради сохранения заказа для своего коллектива разработчиков. В таких случаях ясное изложение идей часто подменяется тяжеловесным жонглированием малопонятными аббревиатурами. Если вы на такой презентации представляете руководство, то вам чрезвычайно важно отличать благие пожелания от реалистичного планирования. Визуальное качество презентации не гарантирует такого же качества у проекта. Более того, зачастую организации, сосредоточенные на качественном выполнении проектов, проигрывают тем, кто меньше беспокоится о реальном производстве эффективных программ.

Наконец, у всех систем существуют такие свойства, которые не представимы классами. Например, надежность, производительность и удобство тестирования не отвечают никаким классам программы, хотя их и можно измерить. Такие свойства систем можно закладывать в проект и измерять в процессе работы программных продуктов. Забота об этих свойствах должна красной нитью проходить через все классы, и ее можно отразить в проектировании и реализации классов и компонентов (§23.4.3).

23.4.3.2. Этап 2: определение операций

Уточните классы, определив набор их операций. Конечно, невозможно полностью отделить выявление концепций и классов от определения операций над ними. Но в процессе поиска классов мы сосредотачиваемся в первую очередь на сущности концепций без упора на операции, в то время как на втором этапе уже можно сосредоточиться именно на операциях, определяя их полный и удобный набор. Действительно, трудно было бы делать это одновременно, особенно в случае параллельной разработки многих, связанных друг с другом классов. Здесь полезными могут оказаться CRC-карточки (§23.4.3.1).

Для выяснения набора необходимых операций могут оказаться полезными следующие соображения:

1. Рассмотрите, как объекты классов должны создаваться (конструироваться), копироваться и уничтожаться.
2. Определите минимальный набор операций, который требует отражаемая классом концепция. Как правило, эти операции реализуются функциями-членами (§10.3).
3. Подумайте, какие операции можно добавить для удобства. Включите только несколько наиболее важных операций. Часто такие операции реализуются в виде *вспомогательных функций* (*helper functions*) (§10.3.2).
4. Подумайте, какие операции должны быть виртуальными, то есть для которых класс обеспечивает интерфейс, а реализацию предоставляет производный класс.
5. Подумайте, какой общности именования и функциональности можно добиться для всех классов компонента.

Это — манифест минимализма. Ясно, что легче добавить, не думая, все функции, которые когда-либо могут понадобиться, и сделать их виртуальными, но чем больше функций, тем более вероятно, что они останутся неиспользованными и только затруднят реализацию системы и ее дальнейшие модификации. В частности, функции, которые непосредственно читают или записывают некоторую часть состояния объекта класса ограничивают класс единственной стратегией развития и сильнейшим образом снижают потенциал развития системы. Такие функции понижают уровень абстракции от концепции до конкретного варианта реализации. Большое количество функций лишь добавляет работы реализующему код программисту и проблемы проектировщику системы при ее дальнейшем совершенствовании. *Гораздо лучше* добавить функцию в тот момент, когда четко выявилась необходимость в этой функции, чем удалять функцию при осознании ее ненужности.

Нужно принимать решение о виртуальности функции самым что ни на есть явным образом (а не в виде детали реализации), ибо это критически влияет на применение класса и на его взаимосвязи с другими классами. Объекты класса всего лишь с одной виртуальной функцией имеют нетривиальную структуру по сравнению с объектами таких языков, как C или Fortran. Класс с единственной виртуальной функцией потенциально может действовать как интерфейс к еще не определенному классу, а виртуальные функции подразумевают зависимость от еще не определенных классов (§24.3.2.1).

Отметим также, что *минимализм* на самом деле *требует от проектировщика больше работы*, а не меньше.

При выборе операций важно сосредоточиться на том, что нужно сделать, а не на том, как это сделать. То есть нам нужно сосредоточиться на желательном поведении, а не на деталях реализации.

Бывает полезно классифицировать операции с точки зрения того, как они используют внутреннее состояние объектов:

- *Фундаментальные операции*: конструкторы, деструкторы и операции копирования.
- «*Инспекторы*»: операции, не модифицирующие состояние объектов.
- *Модификаторы*: операции, модифицирующие состояние объектов.
- *Преобразования*: операции, производящие объект другого типа, отталкиваясь от значения (состояния) объекта, к которому они применяются.
- *Итераторы*: операции, служащие для доступа к последовательности объектов, хранящихся в контейнере.

Перечисленные классификационные категории не ортогональны друг другу. Например, итератор может быть также инспектором или модификатором. Эти категории являются просто классификацией, которая помогла многим людям совершенствоваться в проектировании интерфейсов. Естественно, возможны и другие классификации. Такие классификации помогают согласовывать классы компонента.

Язык C++ помогает отличать инспекторов от модификаторов за счет поддержки константных и неконстантных функций-членов. Аналогично, непосредственно в самом языке поддерживаются конструкторы, деструкторы, операции присваивания и операции приведения (преобразования) типов.

23.4.3.3. Этап 3: выявление зависимостей

Уточните классы в плане их зависимостей. Различные виды зависимостей обсуждаются в §24.3. В контексте проектирования ключевыми являются зависимости параметризация, наследования и использования. Для каждой из них важно ответить на вопрос о том, что означает, что класс ответственен за какое-то одно из свойств системы. Чтобы классу быть ответственным за что-то, ему не обязательно самому содержать все необходимые для этого данные, или непосредственно выполнять все требуемые операции. Более того, классы с фиксированной ответственностью за отдельные свойства системы чаще всего переправляют идущие на них запросы другим классам, которые отвечают за решение частных подзадач. Естественно, тут можно переборщить и создать чрезвычайно неэффективную и труднопонимаемую архитектуру, в рамках которой никакая работа не делается иначе, как порождением каскада обращений объектов за сервисом друг к другу. То, что можно сделать на месте, должно делаться сразу на этом месте.

Необходимость рассмотрения вопросов наследования и отношений использования на этапе проектирования (а не реализации) вытекает из перехода от простого использования классов к представлению концепций задачи. Отсюда следует, что *единицей проекта является компонент* (§23.4.3, §24.4), а *не индивидуальный класс.*

Параметризация — часто приводящая к применения шаблонов — это способ явного представления неявных зависимостей, позволяющая представить альтернативы без введения новых концепций. Часто имеется возможность выбора между тем, чтобы оставить что-то зависеть от контекста, или представить в виде ветви дерева наследования, или воспользоваться параметром (§24.4.1).

23.4.3.4. Этап 4: определение интерфейсов

Определите интерфейсы. Закрытые (private) функции обычно не рассматриваются на этапе проектирования. Вопросы реализации на этапе проектирования по большей части связаны с зависимостями, возникшими на этапе 3. Кроме того, я вывел эмпирическое правило, гласящее: «если для класса не возможны по крайней мере две реализации, то с этим классом что-то не так». Это, скорее всего, не представление концепции, а замаскированная определенная реализация. Во многих случаях, рассмотрение возможности медленной эволюции класса есть приближение к ответу на вопрос «достаточно ли независим интерфейс класса от реализации?».

Открытые базовые классы, а также друзья класса являются частью интерфейса; см. также §11.5 и §24.4.2. Обеспечение раздельных интерфейсов для наследования и для обычных клиентов путем предоставления защищенных и открытых интерфейсов является упражнением, заслуживающим поощрения.

На данном шаге выявляются и специфицируются точные типы аргументов. В идеале нужно стремиться к тому, чтобы максимально полно статически типизировать интерфейсы в рамках типов прикладной задачи; см. §24.2.3 и §24.4.2.

В процессе определения интерфейсов выявляйте классы, чьи операции, как кажется, поддерживают несколько уровней абстракции. Например, некоторые функции-члены класса ***File*** могут принимать аргументы типа ***File_descriptor***, в то время как другие — строковые описания имен файлов. Операции, принимающие ***File_descriptor***, и операции, принимающие имена файлов, работают на разных уровнях абстракции, так что может возникнуть вопрос, относятся ли они к одному

и тому же классу. Может лучше организовать два класса, каждый из которых поддерживает либо файловый дескриптор, либо имена файлов. В типичном случае, все операции класса должны поддерживать одинаковый уровень абстракции. Если это не так, то стоит подумать о реорганизации этого класса и родственных ему классов.

23.4.3.5. Реорганизация иерархии классов

На этапах 1 и 3 мы изучаем структуру классов и классовую иерархию с целью убедиться, что они адекватны нашим нуждам. В общем случае 100%-ой адекватности нет, и мы вынуждены заняться реорганизацией этих структур с целью их улучшения.

Наиболее общими типами реорганизации классов являются вычленение общих частей двух классов и вынос этой общей части в новый класс, или разделение старого класса на два новых класса. В обоих случаях мы приходим к трем классам: одному базовому и двум производным. Когда такую реорганизацию следует делать? Каковы признаки того, что такая реорганизация принесет пользу?

К сожалению, простых и однозначных ответов на эти вопросы нет. Ничего удивительного, ибо мы ищем ответы не на частные вопросы о мелких деталях реализации, а на вопросы об изменениях базовых концепций системы. Здесь самая нетривиальная задача заключается в том, чтобы найти общность между классами и вычленить их общую часть. Точного критерия общности не существует, но этот критерий в любом случае должен затрагивать концепции, например, общность моделей использования, сходство наборов операций, сходство реализаций и даже то обстоятельство, что оба класса одновременно рассматриваются при обсуждении проекта. И наоборот, класс может стать кандидатом для расщепления на два класса, если разные подмножества операций этого класса имеют разные модели использования, если они имеют доступ к разным частям представления данных и даже если по поводу класса возникают разные дискуссии. Заметим также, что иногда превращение схожих классов в шаблон является систематизированным способом предоставления альтернатив (§24.4.1).

Часто проблемы с организацией классовых иерархий возникают из-за плохого именования классов и вытекающих отсюда трудностей в использовании этих имен в процессе обсуждения проекта. Действительно, если при обсуждении проекта ощущается, что имена из иерархии классов звучат нелепо, имеется необходимость и потенциальная возможность организовать иерархию по-другому. Заметьте, что тем самым я хочу сказать, что два человека при обсуждении и анализе иерархии классов намного лучше, чем один. В последнем же случае нужно попытаться составить письменное руководство по проекту с явным выписыванием всех имен классов иерархии.

Одна из самых важных задач проектирования — предоставить интерфейсы, которые могут оставаться стабильными при изменении самих классов (§23.4.2). Часто для этого нужно класс, от которого зависят множество других классов и функций, делать абстрактным классом, предоставляющим самые общие операции. Детали лучше оставить более специализированным производным классам, от которых зависят существенно меньше классов и функций. Чем больше классов зависят от данного класса, тем более общим он должен быть и тем меньше деталей он должен раскрывать.

Существует сильный соблазн — добавлять и добавлять все новые операции (и данные) в класс, используемый многими. Часто это выглядит, как попытка сделать класс более полезным и менее подверженным необходимости в будущих модификациях. Результатом такого образа мыслей является класс с так называемым жирным интерфейсом (§24.4.3) и с полями данных, поддерживающими слабо связанные между собой функции. Это опять-таки подразумевает, что при значительных изменениях в зависимых классах, этот класс также придется переделывать. А это порождает необходимость изменений и в других пользовательских и производных классах. Вместо усложнения основного класса проекта лучше делать его как можно более общим и абстрактным. Особые возможности при необходимости могут быть предоставлены производными классами. Примеры см. в [Martin, 1995].

При этом порождается иерархия абстрактных классов, в которой классы, более близкие к вершине иерархии, являются более общими, и от которых зависят большинство других классов и функций системы. Классы-листья представляют собой самые специализированные классы, и от них непосредственно зависит лишь малая толика кода. В качестве примера рассмотрите окончательную версию иерархии ***Ival_box*** (§12.4.3, §12.4.4).

23.4.3.6. Использование модельных образцов

Собираясь написать статью, я обычно стараюсь подыскать подходящую модель, которой можно было бы следовать. То есть вместо того, чтобы сразу набирать текст, я просматриваю предыдущие статьи на эту тему и прочие публикации, чтобы выбрать из них что-нибудь, что можно взять за начальный образец будущей статьи. Если выбранным образцом (моделью) является моя собственная статья на схожую тему, я могу даже оставить в ней многое в неизменном виде, что-то подправить и добавить лишь что-то новое только там, где этого требует логика новой статьи. Например, данная книга написана на основе первого и второго (предыдущих) изданий. Крайней степенью такой формы работы являются шаблоны писем — я просто заполняю имя и, может быть, добавляю пару строк, чтобы персонализировать послание. По сути дела, в таких письмах я лишь вношу необходимые отличия от стандартного модельного образца.

Такое *использование предыдущих систем* (наработок) в качестве моделей для новых разработок *есть скорее правило*, чем исключение в *любых видах творческой деятельности*. Всюду, где только возможно, проектирование и программирование также должны опираться на ранее выполненные работы. Это позволяет дизайнеру сузить «горизонты» (фронт работ) и сосредоточиться лишь на нескольких новых вопросах. Начинать новый проект с чистого листа — это может быть и вдохновляющий старт, но при попытке все точно специфицировать вы попадаете в ловушку множества альтернатив проектирования, среди которых блуждаете как пьяный. Отталкиваться от существующей модели не столь уж и ограничительно, так как не требуется, чтобы вы перед ней раболепствовали; она лишь освобождает дизайнера от необходимости думать обо всем сразу и позволяет ему рассматривать по одному аспекту системы за раз.

На самом деле, любой проект неявным образом использует модели (предыдущий опыт задействованных проектировщиков). Но явный сознательный выбор конкретной модели делает явными предпочтения, предположения, набор терминов, предоставляет начальную среду разработки и увеличивает вероятность того.

что разные проектировщики будут работать согласованно на базе одного и того же подхода.

Естественно, сам *выбор модели является ответственным проектным решением* и его следует делать только после тщательного поиска, изучения и отбора имеющихся альтернатив. Более того, во многих случаях модель подходит только в том случае, когда необходимые изменения соответствуют лишь адаптации основных идей к специфике нового приложения. Проектирование программ — сложная работа, и нам нужна любая помощь, которую только можно получить: мы не должны отвергать применение моделей из-за неуместного презрения к «подражательству». Подражание (имитация) есть наиболее искренняя форма лести, и применение моделей и результатов ранее выполненных работ для вдохновения (конечно, без нарушения авторских прав и прав собственности) является допустимым приемом творческой работы во всех областях: что было хорошо для Шекспира, то хорошо и для нас. Иногда такое применение моделей в проектировании называется *повторным использованием проектов* (*design reuse*).

Документирование общих элементов множества проектов совместно с описанием решаемых ими проблем проектирования, а также условий, в которых они применимы, является отличной идеей. Задокументированные таким образом наиболее общие и особо полезные элементы проектов называют *образцами* или *паттернами* (*patterns*), и существует литература по документированию образцов и их использованию (например, [Gamma, 1994] и [Coplien, 1995]).

Проектировщику неплохо познакомиться с наиболее известными проектными образцами его предметной области. Как программист, я предпочитаю такие проектные образцы, которые сопровождаются примерами кода. Как и большинство людей, я лучше понимаю основную идею (в данном случае это проектный образец), когда я имею конкретный пример ее применения (здесь это фрагмент кода, иллюстрирующий используемый образец). Люди, интенсивно использующие проектные образцы, применяют специальную терминологию, помогающую им точнее понимать друг друга. К сожалению, это же может превратиться в закрытый язык, отсеивающий непосвященных, а ведь, как всегда, очень важно установить тесное общение между разными людьми, участвующими в проекте (§23.3) и, в первую очередь, между проектировщиками и программистами.

Каждый успешный программный проект является результатом перепроектирования некоторых работающих программных систем меньшего размера. Я не знаю исключений из этого правила. Можно привести множество примеров неудачных проектов, над которыми работали годами, потратили огромные средства, и, наконец, добились успеха, но много позже первоначальных сроков. Как правило, в таких проектах сначала строилась (непреднамеренно и неосознанно) неработающая система, которая затем трансформировалась в систему работающую, и, наконец, эта система перепроектировалась и реорганизовывалась так, что начинала удовлетворять первоначально заявленным целям. Это говорит о том, что наивно пытаться разрабатывать большие системы «с нуля», лишь следуя новейшим принципам. Чем больше и амбициознее проект, за который вы беретесь, тем важнее для него подобрать исходную модель, с которой можно спокойно начать работу. Для больших систем в качестве моделей следует выбирать нечто меньшее, но уже работающее.

23.4.4. Экспериментирование и анализ

Начиная разрабатывать крупный амбициозный проект, мы не знаем лучшего способа его построения. Часто мы даже не до конца понимаем, что именно система должна делать, поскольку частности проявляются лишь после построения системы, ее тестирования и эксплуатации. Как же нам все-таки получить информацию, необходимую для понимания того, какие проектные решения важнее всего для разрабатываемой системы, и какие последствия могут быть у проектных решений?

Нужно проводить эксперимент. Нужно сразу же анализировать проект и реализацию, как только мы получаем хоть что-то для анализа. Часто большую пользу дает обсуждение альтернативных проекта и реализации. Во всех случаях, кроме редчайших, проектирование — это коллективная деятельность, в рамках которых проекты представляются и анализируются. Часто важнейшим инструментом визуализации концепций программы является классная доска — без нее они пребывают втуне.

Похоже, что самая распространенная форма эксперимента состоит в построении *прототипа* (*prototype*), то есть уменьшенной версии всей системы или некоторой ее части. Прототипу не предъявляют жестких требований к быстродействию и ресурсам системы (они полагаются априорно достаточными). В результате в кратчайшие сроки получается работающая версия, с которой можно исследовать проект более предметно и выбирать разные реализации.

При хорошем исполнении этот подход может принести много пользы. Его также можно использовать и для оправдания небрежности. Трудность состоит в том, что цель прототипа может сместиться от исследования проектных альтернатив просто к получению «хоть какой-то работающей реализации системы как можно скорее». В результате может потеряться интерес к внутренней структуре прототипа («в конце концов, это всего лишь прототип») и возникнуть пренебрежение к проектным усилиям, когда все сведется к возне вокруг реализации прототипа. Серьезная опасность таится в том, что все может выродиться в худший образец пожирателя ресурсов и кошмар для сопровождения, при этом создавая иллюзию «почти завершенной» системы. По определению, прототип не имеет ясной внутренней структуры, эффективности и инфраструктуры сопровождения, допускающих его разумное постепенное масштабирование до уровня конечного продукта. Поэтому прототип, превращенный в конечный продукт, лишь впустую отнимает время и энергию у разработчиков. Существует соблазн для проектировщиков и менеджеров превратить прототип в конечный продукт и оставить вопросы эффективности до следующего выпуска. Столь порочная тактика перечеркивает все принципы проектирования.

Другая проблема состоит в том, что разработчики прототипа влюбляются в использованные ими ограниченные приемы программирования и забывают, что этого не может позволить себе вся группа программистов и проектировщиков, и что свобода их маленькой группы от ограничений и формальностей не выдержит столкновения с суровой реальностью жестких временных графиков и взаимозависимостей внутри большой группы разработчиков.

Тем не менее, прототипы могут принести неоценимую пользу. Рассмотрим, например, проектирование пользовательского интерфейса. В этом случае внутренняя структура тех частей системы, которые напрямую не взаимодействуют с пользователем, действительно не важна, и на первый план выступает желание как можно

быстрее прототипировать действующую систему, позволяющую наглядно отработать детали взаимодействия с пользователем. Другой пример — прототип, призванный оказать практическую помощь в исследовании и отработке внутренней структуры программы. Тут уже не важен истинный пользовательский интерфейс и его можно смело отбросить, заменив минимально достаточными рудиментарными средствами имитации взаимодействия с пользователем.

Прототипирование является разновидностью экспериментирования. Желаемым результатом прототипирования является проникновение в проблему в процессе построения прототипа, а не сам по себе прототип. Может быть, самым важным критерием хорошего прототипа является то, что он должен оставаться настолько незавершенным, чтобы было абсолютно очевидно, что это некий экспериментальный аппарат, который просто невозможно превратить в готовый продукт без серьезной переделки проекта и реализации. Оставляя прототип в таком незавершенном состоянии, мы поневоле фокусируемся на эксперименте и минимизируем опасности, связанные с превращением его в конечный продукт. Это также минимизирует искушение делать окончательный проект продукта слишком похожим на проект прототипа, забывая о присущей ему ограниченности. После использования прототип нужно выбросить.

Также следует помнить, что во многих случаях существуют методы экспериментирования, альтернативные прототипам. Ими следует пользоваться во всех случаях, когда есть такая возможность, ибо эти альтернативные методы строже прототипов и налагают меньше нагрузки на проектировщика и на ресурсы системы. В качестве примера можно упомянуть математические модели и разного вида симуляторы. На самом деле, можно представить себе практически непрерывный переход от математической модели через все более подробные имитации, через прототипы, через частичные реализации к завершенной системе.

Отсюда возникает идея о «выращивании» системы путем последовательных повторяющихся переходов от исходного проекта к последующим его стадиям посредством перепроектирования и повторного программирования. Это идеальная стратегия, но она может предъявить слишком большие требования к качеству средств проектирования и реализации. На сегодня эта стратегия применима в случае умеренных по размеру проектов, в которых кардинальные изменения структуры проекта маловероятны, а также для перепроектирования уже выпущенных программных продуктов, для которых означенная стратегия просто неизбежна.

В дополнение к экспериментированию ради понимания возможных способов проектирования актуально исследование особенностей существующего проекта и реализации ради выявления путей дальнейшего совершенствования системы. Например, это может быть исследование зависимостей между классами (§24.3), в котором не следует пренебрегать такими традиционными инструментами разработчика, как графы вызовов функций, замер производительности и т.д.

Отметим, что проект и проектные спецификации столь же подвержены ошибкам, что и реализация. И по сути дела даже больше подвержены, ибо они менее конкретны, определенно менее точны, их нельзя исполнить и, как правило, они не поддержаны столь изощренными средствами, которые доступны для тестирования и анализа реализации. Увеличение уровня абстракции (формализации) языка/обозначений для изложения проекта, способствует применению специальных программных инструментов для поддержки проектирования, что может в определен-

ной степени помочь проектировщику. Но нужно следить за тем, чтобы это не привело к обеднению средств языка программирования, применяемого в реализации (§24.3.1). К тому же, формализованные обозначения сами по себе могут стать источником путаницы и ошибок. Это часто происходит тогда, когда используемый формализм в принципе плохо согласуется с природой решаемой задачи, когда строгость формализма превышает математическую подготовку и опыт проектировщиков и программистов, и когда теряется связь формального описания системы с реальной системой.

Проектированию неизбежно свойственны ошибки. Его трудно поддерживать автоматизирующими инструментами. Главное в проектировании — это опыт и непрерывная учеба на результатах. Поэтому абсолютно неправильно рассматривать разработку программного продукта как линейный процесс, начинающийся анализом и заканчивающийся финальным тестированием. Важно сфокусироваться на итеративной природе проектирования и реализации, чтобы получить максимальную отдачу от обратной связи и опыта предыдущих итераций.

23.4.5. Тестирование

Можно считать, что если программа не тестировалась, то она не работает. Идеальное проектирование и такого же качества реализация системы, в результате которых программа абсолютно корректно работает с самого первого запуска, достижимы разве что в самых тривиальных случаях. Стремиться к идеалу надо, но нужно не забывать о необходимости тестирования.

«Как тестировать?» — на этот вопрос нет общего ответа. А вот на вопрос «Когда тестировать?» общий ответ существует: как можно раньше (и как можно чаще). Стратегия тестирования должна быть частью общего проекта или хотя бы разрабатываться параллельно с ним. *Как только система может быть запущена, следует начинать ее тестирование.* Откладывать серьезное тестирование до момента, когда будет получена почти окончательная реализация — значит обрекать систему на серьезные изъяны, а сроки разработки проекта на срыв.

Если только это вообще возможно, систему нужно *проектировать с прицелом на удобство тестирования*. В частности, механизмы, облегчающие тестирование, могут встраиваться в саму систему. Часто этого не делают из опасений перегрузить систему и что средства, необходимые для тестирования, чрезмерно расширят структуры данных. Такие опасения совершенно необоснованны, ибо дополнительный тестировочный код и расширения структур данных могут быть удалены перед выпуском окончательной версии. Здесь особо удобен механизм *проверочных утверждений* (*assertions*) (§24.3.7.2).

Но еще более важным, чем конкретные тесты, является разработка такой структуры системы, в рамках которой можно быть уверенным в том, что ошибки будут устранены путем комбинации статических проверок, статического анализа и тестирования. Когда разрабатывается общая стратегия устойчивости к ошибкам (§14.9), стратегия тестирования может разрабатываться как дополнительный и тесно связанный с ней аспект общего проектирования.

Если на стадии проектирования вопросы тестирования были забыты, то готовьтесь к тому, что процесс тестирования затянется, сроки сдачи продукта сорвутся, а поддержка продукта усложнится. Обычно хорошим местом для начала работы над

стратегией тестирования является этап проектирования, связанный с определением интерфейсов классов и их зависимостями (§24.3, §24.4.2).

Обычно трудно сказать, каков достаточный объем тестирования. На практике недостаточное тестирование встречается гораздо чаще избыточного. Точное количество ресурсов (в том числе человеческих), которые должны быть выделены под тестирование по сравнению с проектированием и реализацией, зависят от природы задачи и от методов, которыми ее решают. Однако в качестве эмпирического правила я могу предположить, что *на тестирование нужно выделить больше ресурсов* (времени, талантов и т.д.), *чем на ее первоначальную реализацию*. Тестирование должно сосредотачиваться на ошибках, влекущих за собой катастрофические последствия, а также на менее серьезных проблемах, но которые встречаются чаще других.

23.4.6. Сопровождение и поддержка программ

«Сопровождение (поддержка) программ» — это фраза, вводящая в заблуждение. Слово поддержка вызывает аналогию с обслуживанием техники. Но программу не нужно смазывать, заменять в ней износившиеся части, удалять застрявшую в полостях воду, вызывающую ржавчину. Программные продукты можно абсолютно точно реплицировать и перемещать на гигантские расстояния за минимальную цену. Программы — это не техника (аппаратура, железо).

Под сопровождением программ понимается деятельность, направленная на перепроектирование и повторную реализацию; таким образом, она вполне укладывается в обычный цикл разработки программ. Когда гибкость, расширяемость и возможность переноса на иные платформы закладываются в проект с самого начала, серьезных проблем с сопровождением не возникает.

Как и в случае тестирования, рассмотрение вопросов сопровождения системы не следует откладывать на потом и их не нужно отделять от всей разработки в целом. В частности, важно сохранять некоторую преемственность в коллективе разработчиков системы. Нелегко передать вопросы сопровождения готовой системы коллективу, который не принимал участия в ее разработке и не имеет связи с разработчиками. Когда смена персонала неизбежна, нужно особо внимательно отнестись к передаче знаний о структуре системы и ее особенностях (целях). Если группа поддержки окажется в ситуации, когда ей самой придется гадать о целях системы и о ее внутреннем устройстве, исходя из кода реализации, структура системы может стремительно ухудшиться под натиском локальных исправлений. Обычно тут мало помогает документация, ибо последняя в основном нацелена на освещение деталей, чем на раскрытие ключевых идей и принципов функционирования системы.

23.4.7. Эффективность

Дональд Кнут заметил, что «слишком ранняя оптимизация — корень всех зол». Некоторые люди усвоили этот тезис слишком буквально и считают злом любую заботу об эффективности. Разумеется, это не так, и вопросы эффективности нельзя упускать из виду в процессе проектирования и реализации. Но проектировщик не должен рассматривать приемы достижения микроэффективности, а должен сосредотачивать свое внимание на высокоуровневых аспектах эффективной работы системы.

Наилучшая стратегия достижения эффективности — это создание ясного и не переусложненного проекта. Только такие проекты остаются стабильными на протяжении жизни системы и служат базой для повышения ее быстродействия. Не нужно добавлять в проект все новые возможности «просто на всякий случай» (§23.4.3.2, §23.5.3) — это заканчивается удвоением или утроением первоначального размера и сопоставимым снижением быстродействия. Такие переусложненные системы трудно анализировать, так что становится нелегко понять, где можно избежать затрат, а где нельзя. В результате пропадает желание анализировать и повышать эффективность. Оптимизация должна быть результатом тщательного анализа и замеров производительности, а не результатом мелкой возни с кодом программы. Особенно в больших программах интуиция проектировщика или программиста не является надежным средством повышения производительности.

Конечно, важно избегать в программе особо неэффективных конструкций или таких конструкций, для достижения эффективной работы которых потребуется много времени и усилий. Аналогично, важно минимизировать применение плохо переносимых конструкций и средств, поскольку при этом проект завязнет на старых аппаратных средствах, снижая, тем самым, результирующую эффективность работы системы.

23.5. Отдельные аспекты управления проектами

Большинство людей делают в первую очередь то, за что их поощряют. В частности, если кто-то получает при разработке проекта определенные рычаги управления работой других людей, то лишь редкостные личности будут, рискуя карьерой, до конца отстаивать свои взгляды на то, что они считают правильным, перед лицом оппозиции, равнодушия и бюрократии. Отсюда следует, что организации должны вводить систему поощрений, соответствующую заявленным целям проектирования и программирования. Серьезного изменения в программировании можно добиться лишь в связи с серьезным изменением стиля проектирования, а все это достигается серьезными изменениями в стиле управления проектом. Интеллектуальная и организационная инерция способствуют осуществлению сугубо локальных изменений, в то время как для серьезного успеха обычно нужны глобальные изменения. Типичным примером является ситуация, когда осуществляется переход на язык объектно-ориентированного программирования, такой как C++, но при этом по-прежнему используются старые проектные стратегии, не способствующие эффективному использованию новых средств языка (§24.2). Другой пример — переход на объектно-ориентированное проектирование при сохранении старого (не объектно-ориентированного) языка программирования.

23.5.1. Повторное использование кода

Увеличение степени повторного использования готовых проектов и программного кода часто называют в качестве главной причины перехода на новый язык программирования или новую стратегию проектирования. Но на практике часто бывает наоборот — организации скорее поощряют программистов «изобретать велосипед». Например, производительность программиста замеряют по числу строк кода; будет он после этого стремиться к фрагментам малого размера, активно ис-

пользующим стандартную библиотеку, за счет снижения собственного дохода и, возможно, статуса? Менеджеру могут платить в зависимости от количества людей в его подразделении; будет он после этого стремиться использовать готовые наработки других отделов вместо того, чтобы нанять еще несколько человек в свою группу? Компания может получить финансирование пропорционально величине предполагаемых затрат; будет эта компания стремиться минимизировать свою прибыль за счет применения наиболее эффективных средств разработки, позволяющих уменьшить затраты? Конечно, непосредственно платить за повторное использование проектов и кода нелегко, но если менеджмент не найдет возможности так или иначе поощрять участников проекта за такую экономию сил и средств, никакого повторного использования не будет.

Повторное использование предыдущих разработок — это социальная проблема. Воспользоваться готовыми программными проектами можно лишь тогда, когда:

1. Они работают: чтобы годиться к повторному использованию, они должны на самом деле использоваться.
2. Они понятны: важны программная структура, комментарии, документация и руководство.
3. Они могут сосуществовать с иными программами, специально не предназначенными для совместной работы с ними.
4. Они имеют поддержку.
5. Они экономичны (затраты на их разработку и сопровождение равномерно распределены между многими пользователями).
6. Они доступны.

Можно утверждать, что нельзя твердо считать программный компонент готовым для повторного использования до того момента, как кто-либо реально воспользуется им с этой целью. Задача подгонки компонента к новой среде работы требует, как правило, определенной корректировки его операций, универсализации поведения и улучшения способности сосуществовать с другими программами. Пока все это не будет проделано хотя бы раз, даже компоненты, спроектированные и реализованные с величайшим вниманием, будут «спотыкаться об острые углы».

Мой опыт говорит о том, что условия для реального повторного использования кода возникают лишь тогда, когда кто-либо всерьез возьмется за это дело. В малых коллективах это означает, что кто-то вдруг (намеренно или случайно) начинает хранить общие библиотеки и документацию. В больших организациях для этой цели выделяется отдел, призванный собирать, группировать, документировать, популяризировать и поддерживать программные фрагменты, предназначенные для использования многими другими отделами.

Важность таких групп по «стандартным компонентам» невозможно переоценить. Если организация не имеет механизма, поощряющего кооперацию и повторное использование наработок, то этого и не будет в ее работе. Группа «по стандартным компонентам» обязана пропагандировать эти компоненты, так как одной лишь хорошей документации недостаточно. Кроме того, эта группа должна составлять руководства, из которых потенциальные пользователи могут узнать, где найти компонент и какую пользу он может принести. То есть характер работы такой группы близок характеру работы групп маркетинга и повышения квалификации персонала.

По мере возможности члены группы «по стандартным компонентам» должны работать как можно ближе к разработчикам, ибо только в этом случае они смогут понять нужды своих пользователей и осознать возможности совместной работы разных приложений со стандартными компонентами.

Успех группы «по стандартным компонентам» должен измеряться успехом их клиентов. Если же его измерять количеством средств и инструментов, в полезности которых удалось убедить коллективы разработчиков, то такие группы деградируют и становятся простыми распространителями программных продуктов и поборниками изменений ради изменений.

Не любой код может повторно использоваться, так что способность к повторному использованию не является универсальной характеристикой. Когда говорят, что компонент годится к повторному использованию в рамках заданной среды разработки, это означает, что в этом случае не потребуется много дополнительной работы. В то же время, переход к другой среде разработки может потребовать значительных усилий, так что способность к повторному использованию напоминает свойство переносимости. Важно осознать, что повторное использование — это результат проектирования, направленного на достижение этого свойства, результат усовершенствования компонент на основе опыта их применения и результат сознательных усилий по поиску компонентов, пригодных для повторного использования. Повторное использование не возникает волшебным образом из бездумного применения отдельных языковых черт или технологий кодирования. Такие средства языка C++, как классы, виртуальные функции и шаблоны помогают спроектировать систему так, что ее повторное использование становится более легким (и более вероятным), но сами по себе эти средства ничего не гарантируют.

23.5.2. Масштаб

Отдельные программисты или организации легко поддаются идее «делать все правильно». На специфическом жаргоне это звучит как «разработка в соответствии со строгими процедурами». Но при этом может пострадать здравый смысл, будучи потесненным пылким стремлением улучшить обычный ход дел. К сожалению, когда теряется здравый смысл, ничем не ограничивается тот вред, что наносится неограниченным «новаторством».

Рассмотрим стадии процесса разработки, перечисленные в §23.4, и этапы проектирования, перечисленные в §23.4.3. Относительно легко переделать перечисленные моменты в методику проектирования, где каждый этап точно определен и имеет четко обозначенные вход и выход, а также полуформальное описание этих входа и выхода. Можно разработать список проверочных мероприятий, гарантирующих обязательное применение методики, а также инструменты, помогающие контролировать соглашения по процедурам и формальной нотации. Глядя на классовые взаимосвязи (как они представлены в §24.3), можно оценить некоторые из них как хорошие, а другие — как плохие, но такие оценки неплохо подкрепить специальным инструментом, гарантирующим однородность этих оценок для всего проекта. Для завершения процедуры «укрепления проекта» можно определить стандарты документации (включая орфографические и грамматические правила и способ набора текста) и правила приведения кода к общему виду (включая спецификации применяемых средств языка, применяемых библиотек, соглашений по отступам и по именам функций, переменных, типов и т.д.).

Многое из этого может способствовать успеху проекта. Во всяком случае, безрассудно приступать без четко определенной и достаточно строгой схемы деятельности на основе перечисленных выше правил к разработке системы, которая в результате работы сотни людей в конечном итоге будет содержать порядка десяти миллионов строк кода и которая будет поддерживаться еще большим количеством людей в течении десяти и более лет.

По счастью, большинство систем не попадает в эту категорию. И тем не менее, как только принимается некоторый метод проектирования или декларируется приверженность какому-либо методу кодирования и составления документации как «единственно правильному», это может подвигнуть к требованию применять метод всюду и неукоснительно. Для небольших проектов это может породить нелепые ограничения и бессмысленные затраты. В частности, это может стимулировать излишнее бумаготворчество и заполнение отчетов в качестве оценки работы вместо выполнения более полезной работы. В этом случае настоящие программисты и проектировщики покинут проект, а останутся лишь бюрократы.

Стоит такому нелепому применению (на самом деле вполне обоснованного) метода проектирования показать на практике плохие результаты, как тут же начинают шарахаться в противоположную сторону и полностью отказываться от какой-либо формализации проектов. А это, в свою очередь, порождает хаос и неудачи, предотвращать которые и призван регулярный метод проектирования.

Реальный вопрос состоит в том, как найти оптимальную степень формализации для конкретного проекта. Не ждите простых ответов на этот вопрос. Для маленьких проектов могут работать практически любые подходы. Но что существенно хуже — практически любые подходы используются и для больших проектов — неважно сколь плохи эти подходы сами по себе и по отношению к разработчикам — если есть желание потратить на разработку уйму времени и денег.

Ключевой проблемой каждого программного проекта является забота о сохранении целостности проекта. Эта проблема нелинейно нарастает с ростом размера проекта. Только лишь индивидуальный программист или небольшой коллектив программистов может охватить взором и удерживать в памяти все детали и цели проекта. В общем же случае, приходится тратить столь много времени на подпроекты, технические детали, ежедневное администрирование и т.д., что общепроектные цели забываются или подчиняются более локальным и злободневным целям. Нужно во избежание неудачи выделять одного человека или группу людей со специальной задачей по поддержке целостности проекта. И, конечно, такой группе нужно дать полномочия влиять на весь проект в целом.

Отсутствие долговременной стратегии наносит проекту больше вреда, чем какие-либо другие, более частные недостатки. Работа нескольких людей должна выражаться в формулировании общей цели, в постоянном обсуждении этой цели, в написании ключевых документов по проекту в целом, и в том, чтобы помочь всем остальным участникам проекта не забывать про эту общую цель никогда.

23.5.3. Личности

Описанный метод проектирования выводит на первое место мастерство и квалификацию проектировщиков и программистов, так что их выбор критически важен для успеха всей организации.

Руководители часто забывают, что организация в своей основе состоит из личностей. Распространено мнение, что все программисты более-менее одинаковы и вполне взаимозаменяемы. Это заблуждение может развалить организацию, вытолкнув из нее наиболее эффективных личностей и заставив остальных работать ниже их потенциальных возможностей. Личности взаимозаменяемы лишь тогда, когда им не дают применить свои таланты выше того минимума, который требует решаемая задача. Миф о взаимозаменяемости личностей негуманен и расточителен по своей сути.

Большинство методик оценки производительности труда программиста поощряют расточительность и не в состоянии учесть критически важный вклад конкретной личности. Наиболее очевидным примером является практика оценки труда в количестве строк кода, количестве страниц документации, количестве проведенных тестов и т.д. Такие цифры неплохо смотрятся на диаграммах менеджеров, но к реальности они имеют весьма отдаленное отношение. Например, если производительность измеряется числом строк кода, успешное повторное использование кода оказывает отрицательное влияние на оценку программиста. Успешное применение лучших принципов перепроектирования больших программ производит, как правило, такой же эффект.

Измерить качество работы намного сложнее, чем применить валовой принцип подсчета, но все равно крайне важно попытаться поощрить качественную работу индивидуального программиста или групп программистов, вместо того, чтобы тупо подсчитывать количество строк кода. К сожалению, насколько я знаю, никаких попыток разработки мер оценки качества не ведется. В результате, ограниченные методики измерения выполнения работы и состояния проекта лишь искажают реальное представление о разработке. Люди приспосабливаются к установленным промежуточным срокам и подгоняют под них свою работу. В результате страдают как целостность проекта, так и совокупная производительность труда. Например, если постулируется, что к такому-то сроку нужно устранить столько-то ошибок, то это, скорее всего, будет сделано за счет снижения производительности системы и повышения ее требований к аппаратуре. И наоборот, если за критерий выполнения работы будет положено ее быстродействие (то есть измерениям будет подвергнуто исключительно быстродействие), то результат будет достигнут за счет увеличения числа сопутствующих ошибок. Отсутствие объективных критериев измерения качества налагает высокие требования к технической квалификации менеджеров, ибо в ее отсутствие наблюдается неминуемая тенденция награждать за «бурную деятельность», а не за реальные успехи. Не забывайте, что менеджеры — тоже люди. Им нужны знания и квалификация не меньшие, чем у людей, которыми они управляют.

Как и в других областях человеческой деятельности, при разработке программ требуется видеть перспективу. Трудно оценить отдельного человека на основе его деятельности за один всего лишь год. Однако большинство работников имеют достаточно долгий срок работы, чтобы на этом базисе можно было достаточно уверенно судить об их технической зрелости и средней производительности. Отказ от учета сведений за достаточно долгие периоды рабочей деятельности, как это делается там, где людей считают взаимозаменяемыми винтиками рабочего колеса организации, оставляет менеджеров на милость бессмысленных количественных измерений.

Одно из последствий долгосрочного подхода и отказа от «школы управления взаимозаменяемыми идиотами» состоит в том, что личностям (и разработчикам, и руководителям) приходится дольше врастать в серьезную и ответственную работу. Это отбивает охоту к легкомысленным переходам с одной работы на другую или к смене отделов ради карьерного роста. Важной целью является понижение текучести как среди технического персонала, так и среди ключевых менеджеров. Ни один руководитель не может добиться успеха без взаимопонимания с основными проектировщиками и программистами, а также без обновляемых технических знаний. Аналогично, никакая группа проектировщиков и программистов не сможет добиться успеха в долгосрочной перспективе без поддержки грамотных управленцев и без выхода за технические рамки их деятельности.

Там, где требуются новации, старший технический персонал, аналитики, проектировщики, ведущие программисты и т.д. вынуждены начать знакомиться с новыми для них вещами. Эти люди должны изучать новые технологии и отказываться от старых привычек. Это нелегко. Такие люди, как правило, внесли большой вклад в предыдущие успехи организации, опиравшиеся на старые методы, в которых они достигли большого мастерства, и они приобрели на этом неплохую репутацию. Это же относится и к руководителям.

Естественно, такие люди боятся перемен. Это приводит к переоценке сложностей, сопровождающих внедрение новых методов, и к недооценке трудностей, порождаемых упорным стремлением придерживаться старых методов работы в новых условиях. Так же естественно, что люди, ратующие за перемены, склонны недооценивать трудности перехода к новым методам, и к переоценке проблем, порождаемых старыми приемами работы. Эти группы должны общаться между собой, должны учиться говорить на одном языке и выработать в итоге оптимальную модель перехода. Иначе — организационный паралич и уход наиболее способных личностей из обеих групп. Обе группы должны помнить, что самые ярые консерваторы — это вчерашние ярые радикалы. Получив возможность приобрести новый опыт без унижения, ведущие проектировщики и программисты станут наиболее успешными и разумными сторонниками перемен. Их здравый скептицизм, знание запросов конечных пользователей и умение справляться с организационными барьерами бесценны. Ратующие за скорые и бескомпромиссные изменения должны понять, что переход к новым технологиям требует постепенного привыкания. А те, у кого нет никакого стремления к переменам, должны поискать иные сферы приложения талантов, а не вести арьергардные бои в условиях, когда необходимость перемен объективно диктуется новыми условиями достижения успеха.

23.5.4. Гибридное проектирование

Внедрение новых методов работы всегда болезненно. Потрясение для организации и работающих в ней людей могут быть значительными. В частности, резкие перемены, когда через день опытные разработчики «старой школы» превращаются в неэффективных новичков «новой школы», абсолютно неприемлемы. В то же время, серьезные цели редко когда достигаются без серьезных изменений, которые связаны с определенным риском.

Язык C++ разрабатывался таким образом, чтобы минимизировать этот риск, позволяя постепенно вовлекать новые средства. Очевидно, что наибольших выгод

от C++ можно добиться, применяя абстракцию данных, объектно-ориентированное программирование и объектно-ориентированное проектирование, но не столь очевидно, что скорейший способ добиться этого — радикально порвать с прошлым. Иногда такой резкий переход возможен. Но чаще бывает так, что желание перемен сдерживается заботой о том, как такой переход совершить управляемым образом. Учтите следующие соображения:

Проектировщикам и программистам нужно время для того, чтобы приобрести новые навыки.

- Нужно, чтобы старый код работал совместно с новым кодом.
- Старый код нужно продолжать поддерживать.
- Незавершенные старые проекты нужно завершить (в срок).
- Инструменты для новых технологий должны быть адаптированы в текущую среду разработки.

Все это естественным образом порождает гибридный стиль проектирования и программирования, хотя у разработчиков и не было такого намерения.

Легко недооценить первые два из перечисленных пунктов. Поддерживая разные парадигмы программирования, язык C++ помогает организациям совершить плавный переход к новым методам работы из-за того, что:

- Изучая язык C++, программисты могут одновременно выполнять работу.
- Язык C++ дает значительные выгоды даже в условиях недостаточного инструментального окружения.
- Фрагменты программы на C++ могут отлично сочетаться с фрагментами, написанными на C и других традиционных языках.
- Язык C++ имеет большое подмножество, совместимое с языком C.

Центральная идея заключается в том, что программист может постепенно переходить к C++ от традиционных языков, сохраняя традиционный (процедурный) стиль программирования. Затем можно будет воспользоваться абстракцией данных. И, наконец — когда язык C++ и связанные с ним инструменты будут тщательно изучены — можно переходить к объектно-ориентированному и обобщенному программированию. Отметим, что использовать хорошо разработанные библиотеки намного легче, чем спроектировать их и реализовать, поэтому даже новички могут пользоваться преимуществами абстракции данных уже на самых ранних стадиях в целом длительного перехода.

Возможность постепенного изучения языка C++, объектно-ориентированного программирования и объектно-ориентированного проектирования поддерживается возможностью смешивать код на C++ с кодом на других языках, которые не поддерживают абстракцию данных и объектно-ориентированное программирование (§24.2.1). Многие интерфейсы вполне можно оставить процедурными, так как превращение их во что-нибудь более сложное не даст мгновенных преимуществ. Часто библиотеки так и организованы, и поэтому использующий их программист может оставаться в неведении о настоящем языке реализации библиотеки. Применение библиотек, написанных на C, являлось для языка C++ первой и самой важной формой повторного использования кода.

Следующая стадия (нужная там, где требуется более изощренная техника программирования) — представить процедуры, написанные, например, на C или

Fortran, в виде интерфейсных классов C++, инкапсулирующих структуры данных и функции. Простейший пример такого повышения семантики от «структуры данных плюс функции» к абстрактным данным служит строковый класс из §11.12: там инкапсуляция С-строк и строковых функций стандартной библиотеки языка С позволила создать абстрактный строковый тип, пользоваться которым и проще, и удобнее.

Аналогично, любой пользовательский или встроенный тип можно внедрить в иерархию классов (§23.5.1). Это позволяет проектам на C++ эволюционировать в направлении абстракции данных и классовых иерархий, несмотря на то, что в них используется некоторое количество кода на процедурных языках, не поддерживающих перечисленных концепций, и даже в направлении готового продукта, который можно будет использовать в проектах на процедурных языках.

23.6. Аннотированная библиография

Поскольку настоящая глава лишь поверхностно затрагивает сложнейшие вопросы проектирования программ и управления процессом их разработки, я привожу здесь краткую аннотированную библиографию. Более полную аннотированную библиографию можно найти в [Booch, 1994].

[Anderson, 1990] Bruce Anderson and Sanjiv Gossain: *An Iterative Design Model for Reusable Object-Oriented Software.* Proc. OOPSLA'90. Ottawa, Canada. Описание модели итеративного проектирования на конкретном примере и обсуждением опыта его применения.

[Booch, 1994] Grady Booch: *Object-Oriented Analysis and Design with Applications.* Benjamin/Cummings. 1994. ISBN 0-8053-5340-2. Детальное рассмотрение всех аспектов проектирования вместе с графическими инструментами и практическими примерами, поддержанными кодом на C++. Превосходная книга — настоящая глава обязана ей многим. В ней содержатся более глубокие трактовки вопросов, затронутых в данной главе.

[Booch, 1996] Grady Booch: *Object Solutions.* Benjamin/Cummings. 1996. ISBN 0-8053-0594-7. Описание процесса разработки объектно-ориентированных систем с точки зрения менеджмента. Содержит подробные примеры на C++.

[Brooks, 1982] Frederick P. Brooks, Jr.: The Mythical Man-Month. Addison-Wesley.1982. Переиздана с дополнениями в 1997. ISBN 0-201-83595-9. Эту книгу нужно перечитывать каждую пару лет — прививка от высокомерия. И хотя она немного устарела в техническом плане, это абсолютно не касается человеческого и корпоративного факторов, а также вопросов масштабирования.

[Brooks, 1987] Frederick P. Brooks, Jr.: *No Silver Bullet.* IEEE Computer, Vol. 20, No. 4. April 1987. Суммирующее изложение подходов к разработке программ промышленного масштаба с предостережениями против веры в «серебряную пулю».

[Coplien, 1995] James O. Coplien and Douglas C. Schmidt (editors): *Pattern Languages of Program Design*. Addison-Wesley. 1995. ISBN 0-201-60734-4.

[DeMarco, 1987] T. DeMarco and T. Lister: *Peopleware*. Dorset House Publishing Co. 1987. Одна из немногих книг, которые фокусируются на человеческом факторе в производстве компьютерных программ. Каждый менеджер должен с ней ознакомиться. Удобна для непринужденного чтения. Антибиотик от глупости.

[Gamma, 1994] Erich Gamma, et. al: *Design Patterns*. Addison-Wesley. 1994. ISBN 0-201-63361-2. Практический каталог технологий изготовления гибкого программного обеспечения с широкими возможностями повторного использования. Подробно рассматриваются нетривиальные примеры вместе с кодом на C++.

[Jacobson, 1992] Ivar Jacobson et. al.: *Object-Oriented Software Engineering*. Addison-Wesley. 1992. ISBN 0-201-54435-0. Глубокое практическое описание разработки программ промышленного уровня с упором на примерах использования (use cases). К сожалению, содержит примеры на устаревшем варианте C++ от 1987 года.

[Kerr, 1987] Ron Kerr: *A Materialistic View of the Software «Engineering» Analogy*. In SIGPLAN Notices, March 1987. Рассмотрение разных аналогий в данной и последующих главах основано на этой статье и личных беседах с ее автором.

[Liskov, 1987] Barbara Liskov: *Data Abstraction and Hierarchy*. Proc. OOPSLA'87 (Addendum). Orlando, Florida.
Рассматривается вопрос о том, как с помощью наследования можно скомпрометировать абстракцию данных. Отметим, что стандарт C++ располагает специальными средствами, помогающими избежать подобного рода проблем.

[Martin, 1995] Robert C Martin: Designing Object-Oriented C+ + Applications Using the Booch Method. Prentice-Hall. 1995. ISBN 0-13-203837-4. Показывает, как можно систематическим образом выполнять переход от постановки задачи к коду на C++. Представляет альтернативные решения и принципы выбора между ними. Более практична и более конкретна по сравнению с другими книгами по проектированию. Содержит примеры программ на C++.

[Meyer, 1988] Bertrand Meyer: Object Oriented Software Construction. Prentice Hall. 1988. Стр. 1-64 и 323-334 этой книги содержат хороший взгляд на объектно-ориентированные программирование и проектирование, а также множество здравых практических советов. Остальная часть книги описывает язык Eiffel.

[Parkinson, 1957] C N. Parkinson: *Parkinson's Law and other Studies in Administration*. Houghton Mifflin. Boston. 1957. Одно из самых забавных и язвительных описаний тех проблем, которые несут с собой административные процессы.

[Shlaer, 1988] S. Shlaer and S. J. Mellon *Object-Oriented Systems Analysis* and *Object Lifecycles.* Yourdon Press. ISBN 0-13-629023-X and 0-13-629940-7. Представляет взгляд на анализ, проектирование и программирование, отличающийся от представленного в нашей книге. Однако делает это с применением терминологии, похожей на нашу.

[Snyder, 1986] Alan Snyder: Encapsulation and Inheritance in Object-Oriented Programming Languages. Proc. OOPSLA'86. Portland, Oregon. Возможно, это первое хорошее описание взаимосвязи инкапсуляции и наследования. Содержит также хорошее обсуждение некоторых понятий множественного наследования.

[Wirfs-Brock, 1990] Rebecca Wirfs-Brock, Brian Wilkenson, and Lauren Wiener Designing Object-Orient Software. Prentice Hall. 1990. Описывает метод антропоморфического проектирования, основанного на CRC-карточках. Текст книги имеет склонность к языку Smalltalk.

23.7. Советы

1. Поймите, чего вы хотите добиться; §23.3.
2. Не забывайте, что разработка программ — это человеческая деятельность; §23.2, §23.5.3.
3. Часто доказательство по аналогии — это ошибка (или обман); §23.4.
4. Ставьте четкие и осязаемые цели; §23.4.
5. Не пытайтесь использовать технические приемы для решения проблем, связанных с людьми и обществом; §23.4.
6. Создавая проекты и управляя людьми, старайтесь планировать на долгосрочную перспективу; §23.4.1, §23.5.3.
7. Создавать проекты полезно даже для сколь угодно малых программ; §23.2.
8. В проектировании очень важна обратная связь; §23.4.
9. Не путайте бурную деятельность с реальным прогрессом; §23.3, §23.4.
10. Не пытайтесь обобщать сверх меры, больше того, что можно протестировать и что позволяет ваш опыт; §23.4.1, §23.4.2.
11. Концепции представляйте в виде классов; §23.4.2, §23.4.3.1.
12. У систем имеются свойства, не представимые классами; §23.4.3.1.
13. Иерархические взаимосвязи понятий представляйте в виде иерархии классов; §23.4.3.1.
14. Выискивайте общность в концепциях приложения и представляйте более общие понятия в виде базовых классов; §23.4.3.1, §23.4.3.5.
15. Классификация предметной области не всегда полезна для модели наследования в приложении; §23.4.3.1.

16. Проектируйте классовые иерархии, отталкиваясь от поведения моделируемой системы и от инвариантов; §23.4.3.1, §23.4.3.5, §24.3.7.1.
17. Рассмотрите типичные примеры использования; §23.4.3.1.
18. Рассмотрите возможность использования CRC-карточек; §23.4.3.1.
19. Используйте готовые системы в качестве моделей, вдохновляющих примеров и отправных точек; §23.4.3.6.
20. Остерегайтесь излишней графической инженерии; §23.4.3.1.
21. Отбросьте прототип до того, как он станет помехой; §23.4.4.
22. Проектируйте с учетом неизбежных изменений, обращая внимание на гибкость, расширяемость, переносимость и повторное использование; §23.4.2.
23. Сфокусируйтесь на проектировании компонентов; §23.4.3.
24. Желательно, чтобы каждый интерфейс представлял концепцию на одном уровне абстракции; §23.4.3.1.
25. Проектируйте так, чтобы была определенная стабильность при неизбежности изменений; §23.4.2.
26. Повышайте стабильность проекта, делая интенсивно используемые интерфейсы минимальными, общими и абстрактными; §23.4.3.2, §23.4.3.5.
27. Придерживайтесь принципа минимализма — не добавляйте ничего «просто на всякий случай»; §23.4.3.2.
28. Всегда рассматривайте альтернативные реализации классов. Если альтернатив нет — то, возможно, класс не отражает чистой концепции; §23.4.3.4.
29. Многократно пересматривайте и улучшайте проект и реализацию; §23.4, §23.4.3.
30. Применяйте самые лучшие доступные инструменты для тестирования и анализа проблем, для проектирования и реализации; §23.3, §23.4.1, §23.4.4.
31. Экспериментируйте, анализируйте и тестируйте как можно раньше и как можно чаще; §23.4.4, §23.4.5.
32. Никогда не забывайте об эффективности; §23.4.7.
33. Сопоставляйте уровень формализации масштабу проекта; §23.5.2.
34. Обязательно кто-то должен отвечать за проект в целом; §23.5.2.
35. Документируйте, рекламируйте и поддерживайте компоненты многократного использования; §23.5.1.
36. Документируйте не только детали, но и общие цели и принципы; §23.4.6.
37. В составе документации пишите и руководство для новых разработчиков; §23.4.6.
38. Поощряйте повторное использование проектов, библиотек и классов; §23.5.1.

24

Проектирование и программирование

Стремитесь, чтобы все было просто:
просто, как только возможно, но не проще того.
— А. Эйнштейн

Проектирование и язык программирования — классы — наследование — проверка типов — программирование — что представляют собой классы? — иерархии классов — зависимости — агрегация — агрегация и наследование — альтернативы проектирования —отношение использования — программируемые отношения — инварианты — утверждения — инкапсуляция — компоненты — шаблоны — интерфейсы и реализации — советы.

24.1. Обзор

В этой главе рассматривается, как языки программирования вообще и язык C++ в частности могут поддержать процесс проектирования:

- §24.2 Фундаментальная роль классов, классовых иерархий, проверок типов и собственно программирования.
- §24.3 Применение классов и классовых иерархий с акцентом на взаимозависимости отдельных частей программы.
- §24.4 Понятие *компонента* (*component*), который является основной единицей проекта, и практические соображения по определению интерфейсов.

Более общие вопросы проектирования рассматриваются в главе 23, а различные приемы использования классов более подробно обсуждаются в главе 25.

24.2. Проектирование и язык программирования

Если бы мне пришлось строить мост, я бы серьезно задумался над тем, из какого материала его строить. Ясно, что проект моста в сильнейшей степени зависел бы от выбранного материала, и наоборот. Разумный проект каменного моста отличается от разумного проекта стального моста, деревянного моста и т.д. Я не сумел бы сделать разумный выбор материала без необходимого минимума знаний о материалах и их применении в строительстве. Естественно, что для проектирования деревянного моста не нужно быть искусным плотником, но чтобы осуществить выбор между деревом и сталью, нужно знать основные характеристики деревянных конструкций. Более того, хотя лично вам и не требуется быть искусным плотником при проектировании деревянного моста, вам нужно знать детальные конструкционные характеристики дерева и приемы его обработки (которыми владеют плотники).

Аналогично, чтобы выбрать язык для реализации некоторого программного продукта, надо знать несколько языков программирования, а чтобы спроектировать некоторую часть системы, вам нужно знать выбранный для реализации язык — даже если вам лично и не пришлось писать на нем ни строчки. Хороший проектировщик мостов учитывает свойства применяемого материала и отталкивается от этих свойств для достижения высокого качества проекта. Аналогично, хороший проектировщик программной системы учитывает свойства выбранного языка программирования и старается по максимуму использовать его сильные стороны, а также старается, насколько возможно, избегать приемов его использования, которые вызовут излишние трудности у тех, кто будет реализовывать проект в коде.

Кто-то может подумать, что важность выбора языка актуальна в первую очередь в тех случаях, когда проектировщик/программист — это одно и то же лицо. В этом случае программист может соблазниться не тем языком, как из-за недостатка опыта, так и по причине чрезмерного увлечения некоторыми стилями программирования. Когда же проектируют и программируют разные люди, тогда различия в их опыте, предпочтениях, культуре и знаниях почти наверняка породят ошибки, неэлегантность и неэффективность результирующей системы.

Что может дать язык программирования проектировщику? Он может предоставить средства, позволяющие непосредственно выразить ключевые концепции системы с их помощью. Это упрощает процесс программирования, позволяет легче следить за соответствием между проектом и реализацией, улучшает взаимопонимание между проектировщиками и программистами и позволяет создать более качественные инструменты поддержки, как первых, так и вторых.

Например, большинство методов проектирования концентрируется на взаимозависимостях между разными частями программы (обычно, стараясь минимизировать и точно определить эти зависимости). Язык, позволяющий явным образом выразить интерфейсы взаимодействия между разными частями программы, непосредственно поддерживает такую озабоченность проектировщиков. Он гарантирует, что нет никаких других зависимостей, кроме явно оговоренных. Из-за того, что зависимости явно представлены в коде программы, облегчается создание автоматизированных инструментов, читающих код и строящих наглядные диаграммы зависимостей. Это радикально облегчает работу проектировщиков и всех людей, нуждающихся в изучении структуры программы. Язык C++ — как раз такой язык, который

помогает уменьшить естественный зазор между проектом и программным кодом, и тем самым, уменьшить путаницу и недоразумения.

Ключевое понятие языка C++ — класс. Класс — это тип. Наряду с пространствами имен классы являются механизмом сокрытия информации. Программы могут быть представлены в терминах пользовательских типов данных и их иерархий. И встроенные типы, и типы данных, определяемые пользователем, подчиняются правилам статической проверки типов. Виртуальные функции обеспечивают механизм динамического (позднего) связывания без нарушения правил статической типизации. Шаблоны поддерживают проектирование параметризованных типов. Исключения предоставляют регулярный способ обработки ошибок времени выполнения. И все эти возможности языка C++ не требуют дополнительных затрат памяти и времени по сравнению с программами на языке C. Перечисленные средства относятся к основным средствам языка, которые проектировщик должен хорошо знать и принимать во внимание. Но кроме этого, общедоступные основные библиотеки — библиотеки матриц, интерфейсы к базам данных, библиотеки графического интерфейса пользователя и библиотеки поддержки параллелизма — тоже могут сильно влиять на проектные решения.

Страх перед новизной часто порождает недостаточное использование средств языка C++. К этому же приводит неуместный перенос опыта из других предметных областей, сред разработки и языков программирования. Ухудшить проект могут и неадекватные инструменты проектирования. Стоит упомянуть пять ошибок проектирования, которые не позволяют надлежащим образом воспользоваться преимуществами языка и принять во внимание его ограничения:

1. Отказ от классов — в результате проект приходится выражать с помощью подмножества C++, совпадающего с возможностями языка C.
2. Отказ от производных классов и виртуальных функций — в результате проект использует лишь абстракцию данных.
3. Игнорирование всех возможностей статической проверки типов — в результате программисты вынуждены моделировать динамическую проверку типов.
4. Отказ от самостоятельного этапа программирования — в результате проект приходится подгонять под жесткие рамки, позволяющие обойтись без программистов.
5. Игнорирование возможностей, альтернативных или дополнительных по отношению к классовым иерархиям.

Перечисленные ошибки характерны, соответственно, для проектировщиков, которые:

1. Имеют опыт работы с C, case-средствами и структурным проектированием.
2. Имеют опыт работы с Ada83, Visual Basic и активно используют абстракцию данных.
3. Имеют опыт работы со SmallTalk или Lisp.
4. Имеют опыт работы в нетехнических или каких-либо иных узкоспециализированных областях.
5. «Зациклились» на «чистом» объектно-ориентированном программировании.

В каждом случае следует задуматься, правильно ли выбран язык программирования, правильно ли выбран метод проектирования и не ошибся ли проектировщик в использовании инструментов проектирования.

Любое перечисленное несоответствие — это не преступление и его не надо стыдиться. Это просто несоответствие, которое приводит к неоптимальным проектам и лишним проблемам для программистов, которых лучше было бы избежать. Проблемы возникают и у проектировщиков, когда их среда разработки беднее применяемой среды программирования. По мере возможностей лучше избегать таких несоответствий.

Последующие разделы посвящены более подробному анализу перечисленных выше ошибок.

24.2.1. Отказ от классов

Рассмотрим вариант проектирования с отказом от классов. Результирующая программа на C++ будет примерно такой же, как программа на C (или на COBOL), которая получилась бы в рамках того же самого проекта. Можно сказать, что в этом случае проект системы «не зависит от языка программирования», так как программисты вынуждены ограничивать себя общим подмножеством C и COBOL. У этого подхода есть свои преимущества. Например, строгое разделение данных и кода упрощает работу с традиционными базами данных, разработанными как раз для таких программ. Поскольку используется минимальный язык программирования, от программистов требуется минимальная квалификация. Для многих приложений, например тех, что последовательно работают с базами данных, такой образ мыслей вполне оправдан, а разрабатывавшиеся десятилетиями приемы программирования по-прежнему работоспособны.

Естественно, имеются приложения, которые существенно отличаются от программ последовательной работы с записями, например, более сложные интерактивные CASE-системы. Отсутствие поддержки абстракции данных, вызванной отказом от применения классов, принесет в таких случаях чистый вред. Внутренняя сложность системы обязательно взыграет где-нибудь в такой программе, а бедность языковых средств не позволит непосредственно отразить в ней элементы проекта. В результате, программа имеет излишне большой размер, плохую проверку типов, и плохо приспособлена к поддержке автоматическими инструментами.

Временным решением проблемы может быть построение некоторых частных инструментов, поддерживающих понятия проектного метода. Эти инструменты возьмут на себя проверку программы на более высоком уровне абстракции, а также проверку типов для компенсации недостаточных возможностей (намеренно) обедненного языка реализации. В результате, такой метод проектирования становится специализированным закрытым внутрифирменным языком создания программ. Подобного рода языки программирования в большинстве случаев служат плохой заменой для широко распространенных языков программирования общего назначения, поддержанных подходящими инструментами проектирования.

Самая типичная причина отказа от классов — простая инерция. Традиционные языки программирования не поддерживают понятия класса, так что традиционные методики проектирования вынуждены учитывать это обстоятельство. Главной целью проектирования было разбить задачу на ряд процедур, выполняющих заданные

ограниченные действия. Такой стиль, названный в главе 2 *процедурным программированием* (*procedural programming*), в контексте разработки проектов называется *функциональной декомпозицией* (*functional decomposition*). В связи с этим задают вопрос: «А можно ли применять язык C++ в проектах, основанных на функциональной декомпозиции?». Можно конечно, но в таком случае вы неизбежно придете к тому, что язык C++ будет использоваться просто как улучшенный C, и столкнетесь с проблемами, рассмотренными выше. Это может быть приемлемо в переходный период, для уже завершенных систем, а также для тех подсистем, где классы, возможно, не принесут больших выгод (с учетом опыта вовлеченных в проект исполнителей). В более долгосрочной перспективе широкомасштабная политика отказа от применения классов, обусловленная применением функциональной декомпозиции, не совместима с эффективным применением C++ и других языков, поддерживающих абстракцию данных.

Процедурно-ориентированный и объектно-ориентированный взгляды на процесс создания программ различаются фундаментально и, как правило, порождают принципиально разные способы решения одной и той же задачи. Это утверждение верно как для этапа проектирования, так и для процесса непосредственного программирования: вы можете сфокусировать проект либо на выполняемых действиях, либо на представляемых сущностях, но не на том и другом одновременно.

Так почему же стоит предпочесть объектно-ориентированные методики разработки программ традиционным методам, основанным на функциональной декомпозиции? В первую очередь потому, что функциональная декомпозиция приводит к недостаточной абстракции данных. А отсюда уже следует, что проект будет:

- менее приспособлен к изменениям,
- менее совместим с инструментальными средствами,
- менее приспособлен к параллельной разработке,
- менее удобен для поддержки.

Проблема заключается в том, что функциональная декомпозиция вынуждает делать важные данные глобальными, так как она порождает структуру системы в виде дерева функций, из-за чего данные, с которыми работают несколько функций, должны быть глобальными по отношению к этим функциям. Это приводит к тому, что «интересные» данные всплывают все выше и выше к корню дерева по мере того, как все большее число функций нуждается в доступе к этим данным (считаем, что дерево растет от корня вниз). Совершенно аналогично, в иерархии классов с одним корнем имеется тенденция поднимать «интересные» данные и функции ближе к корневому (базовому) классу (§24.4). Однако когда проект фокусируется на спецификациях классов и инкапсуляции данных, эта проблема решается таким образом, что зависимости между разными частями программы делаются явными и хорошо отслеживаются. Еще важнее то, что при этом вообще ослабляется зависимость разных частей программы друг от друга за счет того, что сильнее локализуются обращения к данным.

Тем не менее, все зависит от конкретных обстоятельств, и некоторые задачи лучше решаются посредством набора процедур. Суть объектно-ориентированного подхода вовсе не в том, чтобы в программе не было глобальных функций и не в том, чтобы в ней вовсе не было процедурно-ориентированных частей, а в том, чтобы программа была разделена на части, каждая из которых наилучшим образом соот-

ветствует своей прикладной области. Чаще всего это достигается в процессе проектирования, ориентированного на классы, а не на глобальные функции. Применять процедурный стиль нужно осознано и обосновано, а не просто так, по умолчанию. И классы, и процедуры должны соответствовать специфике задачи, а не жестко навязанному методу проектирования.

24.2.2. Отказ от производных классов и виртуальных функций

Теперь рассмотрим методы проектирования, игнорирующие наследование классов. В этом случае программы просто отказываются использовать одно из самых основных преимуществ C++, хотя и продолжают получать выгоды от C++ по сравнению с C, Pascal, Fortran, COBOL и т.д. Часто в качестве причин такого отказа — кроме инерции — выдвигаются утверждения, что «наследование — это всего лишь деталь реализации», «наследование нарушает сокрытие информации» и что «наследование мешает взаимодействию с другими программами».

Взгляд на наследование как на деталь реализации не учитывает того, что с помощью наследственных иерархий классов можно непосредственно моделировать ключевые взаимоотношения между концепциями и понятиями приложения. Такие взаимоотношения должны также явно прописываться в проекте с тем, чтобы дизайнеры (проектировщики) могли обсуждать их и анализировать.

Есть определенные резоны в том, чтобы исключить наследование из частей программы, взаимодействующих с кодом, написанным на других языках программирования. Это, однако, не является достаточным обоснованием для полного отказа от наследования в иных частях программы. Нужно всего лишь более тщательно определить и изолировать интерфейс взаимодействия с «внешним миром». Аналогично, беспокойства о нарушении сокрытия информации посредством использования наследования (§24.3.2.1) являются причиной для более тщательного проектирования виртуальных функций и защищенных членов (§15.3). Это не причина отказываться от наследования вообще.

В определенных проектах наследование действительно может и не давать существенного выигрыша. Однако для больших проектов политика «никакого наследования» приводит лишь к менее ясным и гибким системам, в которых якобы отсутствующее наследование симулируется при помощи иных языковых конструкций. В дальнейшем сами программисты вынуждены будут применять наследование для улучшения кода, несмотря на отсутствие его в проекте системы. В результате, проектная политика «никакого наследования» приведет лишь к тому, что в итоге наследование будет встречаться в программе несогласованным, лоскутным образом, а единого согласованного проекта всей системы не получится.

24.2.3. Игнорирование возможностей статической проверки типов

Приуменьшение или игнорирование роли статической проверки типов часто обосновывается рассуждениями вроде того, что «типы — это артефакты языка программирования», «естественнее думать об объектах, а не о типах» и «статическая проверка типов вынуждает нас слишком рано задумываться о деталях реализации». На такие «мелочи», как проверка типов, действительно можно не обращать внимание на ранних стадиях проектирования и анализа. В то же время, классы и классовые иерархии также небесполезны на стадии проектирования: они помогают спе-

цифицировать концепции, выяснить их взаимосвязи и уточнить их смысл. По мере продвижения в разработке проекта эта точность все более обретает форму все более точных спецификаций классов и их интерфейсов.

Важно понять, что точно определенные и строго типизированные интерфейсы являются фундаментальным инструментом проектирования. Язык C++ разрабатывался с учетом этого обстоятельства. Строго типизированный интерфейс гарантирует, что согласованные куски кода могут быть откомпилированы и скомпонованы так, что это позволяет им делать обоснованные предположения друг о друге. Выполнение этих предположений гарантируется системой типов. В свою очередь, это уменьшает необходимость в интенсивном тестировании, упрощая тем самым фазу интеграции проекта, над которым работает множество людей. Интеграция не стала важнейшей частью настоящей главы лишь потому, что уже накоплен большой положительный практический опыт интеграции систем со строго типизированными интерфейсами.

Рассмотрим следующую аналогию. В физическом мире мы соединяем друг с другом различные приборы, и кажется, что число стандартов на разъемы соединений бесконечно. Очевидно, что разъемы специально спроектированы так, чтобы было невозможно соединить два устройства в том случае, если они не предназначены для этого, а соединения осуществлялись лишь надлежащим образом. Например, вам не удастся подсоединить электрическую бритву к высоковольтной сети. Если бы вам это удалось, результатом была бы обуглившаяся электробритва или труп ее владельца. Много усилий потрачено на то, чтобы гарантировать невозможность соединения частей аппаратуры, не предназначенных для этой цели. Иногда вместо несовместимых разъемов в аппаратуру встраивают внутреннюю защиту, например предохранители от скачков напряжения. Более высокую степень защищенности обеспечивает сочетание несовместимости разъемов и дополнительной встроенной динамической защиты от непредвиденных отклонений в режимах использования устройств.

В мире программ аналогия почти полная. Статическая проверка типов равносильна совместимости на уровне разъемов, а динамические проверки сродни динамической защите электрических цепей. Как в физическом мире, так и в программном мире, если не сработали оба вида проверок — жди беды. В больших системах нужно использовать оба вида проверок.

Согласно рассмотренной в §23.4.3 схеме этапов проектирования информация о типах возникает на этапе 2 (возможно, после поверхностного рассмотрения на этапе 1) и становится определяющей на этапе 4.

Статически проверяемые интерфейсы являются главной гарантией совместной работы программного кода, разработанного разными группами людей. Документация на интерфейсы (включая точную спецификацию всех вовлеченных типов) служит основным средством общения между разными группами программистов. Такие интерфейсы являются одним из важнейших результатов проектирования системы, и на них фокусируется общение между проектировщиками и программистами.

Игнорирование статической проверки типов при рассмотрении интерфейсов приводит к проектам, затуманивающим структуру программы и откладывающим обнаружение ошибок до стадии выполнения программы. Например, интерфейс можно определить в терминах самоидентифицирующих себя объектов:

```
// Пример, предполагающий динамическую проверку типов, а не статическую:
Stack s;                    // стек может содержать указатели на объекты любых ьтров

void f()
{
  s.push(new Saab900);
  s.push(new Saab37B);

  S.pop()->takeoff();      // отлично: Saab37B - это самолет
  s.pop()->takeoff();      // run-time error: авто не может взлететь
}
```

Недостаточная точность определения интерфейса (функции **Stack::push()**) привела к необходимости динамической проверки вместо статической. Стек *s* предназначен для хранения самолетов, но в коде это явным образом не отражено, так что теперь пользователь должен сам гарантировать соблюдения этого требования.

Более точная спецификация — шаблон плюс виртуальные функции вместо негарантированной динамической проверки — позволяет выявить ошибку на стадии компиляции:

```
Stack<Plane*> s;           // Stack может содержать только указатели на самолеты

void f()
{
  s.push(new Saab900);     // error: Saab900 - не самолет
  s.push(new Saab37B);

  S.pop()->takeoff();      // чудесно: Saab37B - это самолет
  s.pop()->takeoff();
}
```

Этот вопрос рассматривался также в §16.2.2. Разница в производительности для статической и динамической проверок может быть значительной (в 3-5 раз хуже для динамической проверки).

Не следует, однако, бросаться в другую крайность. Исключительно статическими проверками невозможно выявить все ошибки. Например, очевидно, что никакими статическими проверками невозможно защититься от аппаратных сбоев. Посмотрите также пример из §25.4.1, из которого видно, что полная статическая проверка не всегда достижима. Все же нужно стремиться к тому, чтобы большинство интерфейсов для типов уровня приложения проверялись статически; см. §24.4.2.

Иная проблема состоит в том, что идеальный с абстрактной точки зрения проект может породить проблемы из-за того, что не принимает во внимание ограниченные возможности базового средства — языка программирования, в данном случае C++. Например, функция *f*(), которой требуется выполнять операцию **turn_right**() над своим аргументом, может делать это только при условии общности типов ее аргументов:

```
class Plane
{
  // ...
  void turn_right();
};
```

```
class Car
{
  // ...
  void turn_right ( ) ;
} ;

void f(X* p)                // каков должен быть тип X?
{
  p->turn_right ( ) ;
  // ...
}
```

Некоторые языки (вроде Smalltalk или CLOS) позволяют использовать два типа взаимозаменяемым образом (если у них операции одинаковые), связывая их через общий базовый класс и откладывая разрешение имени до выполнения программы. Язык C++ намеренно выполняет это с помощью шаблонов, разрешая имена во время компиляции. Нешаблонные функции допускают разные типы аргументов, только если они неявным образом приводятся друг к другу. Таким образом, в предыдущем примере ***X*** должен быть общим базовым классом для ***Plane*** и для ***Car*** (то есть общим базовым классом для управляемых транспортных средств).

Как правило, примеры, инспирированные идеями из языков, чуждых C++, могут отражаться в нем путем явного отражения неявных допущений. Например, для классов ***Plane*** и ***Car*** в отсутствие у них общей базы можно создать такую классовую иерархию, которая позволит нам все-таки передавать функции *f(X*)* объекты, содержащие ***Car*** или ***Plane*** (§25.4.1), что впрочем весьма непросто. Шаблоны обычно помогают встраивать в код на C++ чуждые для этого языка идеи. Несоответствие проектных идей возможностям C++ обычно порождает неестественный и неэффективный код, который трудно понимать и поддерживать.

Несоответствие техники проектирования языку реализации можно сравнить с дословным переводом с одного естественного языка на другой. Например, английский язык с немецкой грамматикой так же неуклюж, как немецкий с английской, и оба могут оказаться непонятными для человека, владеющего лишь одним из них.

Классы в программе являются конкретными представлениями концепций проекта. Таким образом, делая менее четкой разницу между классами, мы затуманиваем ключевые концепции проекта.

24.2.4. Отказ от традиционного программирования

Программирование весьма дорого и непредсказуемо по сравнению со многими другими видами деятельности, а результирующий код не на 100% надежен. Трудоемкость ручного программирования высока — большая часть задержек серьезных проектов происходит как раз из-за неготовности кода. Почему бы тогда совсем не отказаться от программирования как вида деятельности?

Для многих руководителей избавление от наглых, недисциплинированных, ненадлежащим образом одетых, слишком высокооплачиваемых и слишком увлекающихся техническим жаргоном программистов кажется безусловным благом. Для программистов это может показаться абсурдным. Однако на самом деле имеется возможность избавиться от традиционного программирования в ряде важных прак-

тических областей, автоматически сгенерировав конечный код исходя из одних лишь высокоуровневых спецификаций. Или можно генерировать код, отталкиваясь от показанных на мониторе компьютера изображений геометрических фигур. Последнее типично для проектирования графического интерфейса пользователя, когда с помощью автоматизированных средств можно получить готовый код во много раз быстрее, чем при традиционном ручном программировании. Аналогично, схема базы данных и доступ к этим данным согласно имеющейся схеме намного проще и быстрее запрограммировать автоматически на основе формальных спецификаций, чем писать для этого код на C++ или иных языках программирования общего назначения. Конечные автоматы, которые будут меньше, быстрее и корректнее чем то, что может произвести большинство программистов, могут генерироваться из спецификаций или на основе визуального манипулирования графическими образами.

Подобные приемы хорошо работают в тех областях, где-либо имеется строгий и однозначный теоретический фундамент (математика, конечные автоматы, реляционные базы данных), либо есть готовый каркас, в который можно добавлять (встраивать) небольшие фрагменты кода (графический интерфейс пользователя, моделирование сетей, схемы баз данных). Очевидная польза от такого приема в ограниченных (чаще всего, принципиально ограниченных) областях может навести кое-кого на мысль о том, что полный отказ от традиционного программирования уже не за горами. Это не так. Фундаментальная причина состоит в том, что за полный отказ от программирования придется заплатить тем, что в относительно несложный в частных случаях язык формальных спецификаций придется внести сложность языков программирования общего назначения. И где в таком случае будет выигрыш?

Иногда также забывают, что сам стандартный каркас приложения, позволяющий избавиться от традиционного программирования в какой-либо конкретной предметной области, был спроектирован, запрограммирован и оттестирован традиционным способом. Язык C++ и описанные в нашей книге приемы проектирования и программирования широчайшим образом применяются как раз для разработки подобного рода систем.

В произвольных предметных областях попытка дизайнеров, ради борьбы со сложностью, придерживаться ограниченных высокоуровневых спецификаций приводит к тому, что обедняются используемые средства языка программирования общего назначения и получается ужасный результирующий код. Для улучшения такого кода программисты вынуждены демонстрировать героизм и отказываться от навязанных высокоуровневых моделей.

Лично я не вижу никаких признаков того, что традиционное программирование может быть успешно устранено отовсюду, кроме областей, где имеется твердый математический базис или приемлемые готовые каркасы приложений. Но даже в последнем случае эффективность разработок тормозится и сходит на нет, когда приходится выходить за первоначальные рамки и решать более общие задачи. Ясно, что вредно и полностью отрицать пользу от каркасов и кодогенераторов на основе высокоуровневых спецификаций, и приписывать этим средствам универсальную силу.

Разработка инструментов, библиотек и каркасов является одной из высших форм проектирования и программирования. Построение полезной математической

модели предметной области является одной из высших форм анализа. Наконец, самостоятельная разработка инструментов, библиотек и каркасов, которые будут доступны тысячам людей, есть еще и способ избежать зависимости от единственного инструмента, абсолютная привязанность к которому превратит любого программиста в ремесленника.

Очень важно, чтобы система формализованных спецификаций или библиотека поддержки автоматизированной разработки могли эффективно взаимодействовать с языком программирования общего назначения. Иначе получающиеся каркасы слишком уж ограничительны. Большое преимущество имеют системы, которые из высокоуровневых спецификаций генерируют код на общедоступном языке программирования общего назначения. Собственные внутренние языки программирования являются долгосрочным преимуществом лишь для поставщика такого решения. Также плохи автоматизированные решения, генерирующие слишком низкоуровневый код. В оптимальном случае нужен компромисс между удобством и простотой спецификаций высокого уровня (или графических манипуляций со зрительными образами) и выразительностью и богатством языка программирования общего назначения. Пожертвовать здесь чем-либо одним в ущерб другому, значит пожертвовать интересами пользователей ради интересов непосредственных разработчиков системы. Успех больших разработок, которые всегда являются многоуровневыми и модульными, обуславливается привлечением и гармоничным сочетанием самых разных языков, библиотек, инструментов и технологий.

24.2.5. Применение исключительно классовых иерархий наследования

Когда мы обнаруживаем, что нечто новое действительно хорошо работает, мы теряем чувство меры и применяем это уже без разбору. Другими словами, удачное решение некоторых проблем кажется удачным решением для почти всех проблем. Классовые иерархии и полиморфные операции над их объектами действительно служат удачным решением многих проблем. В то же время, не каждую концепцию задачи можно удачно отобразить некоторой частью иерархии, и не каждый программный компонент реализуем в виде иерархии классов.

Почему? Класс отражает некоторую концепцию, а классовая иерархия отражает взаимоотношение между классами. Теперь скажите, что общего между улыбкой, CD-ROM приводом, записью оперы, строчкой текста, спутником, моей медицинской картой и часами реального времени? Помещение всего этого в единую классовую иерархию, когда общность состоит лишь в том, что все это артефакты программирования (то есть просто объекты), не имеет фундаментального значения и ведет лишь к путанице (§15.4.5). Действительно, загоняя все в единую иерархию, мы создаем искусственное сходство и затушевываем реальное. На самом деле, развивать такие классовые иерархии нужно лишь тогда, когда анализ сущностей приложения выявил фундаментальную общность концепций, или когда в процессе проектирования и программирования открылась полезная общность в применяемых для реализации концепций структурах. В последнем случае нужно четко различать истинную общность (подтипизация с применением открытого наследования) и полезные упрощения реализации (с помощью закрытого наследования; §24.3.2.1).

Такой образ мыслей приводит к программе, содержащей несколько несвязанных или слабо связанных классовых иерархий, каждая из которых представляет множество тесно связанных между собой концепций. Это также приводит к понятию конкретного класса (§25.2), не входящего в иерархию, ибо помещение его в иерархию лишь испортит производительность и нарушит независимость такого класса от остальных частей системы.

Чтобы быть эффективной, важная операция класса, входящего в иерархию, должна быть виртуальной функцией. Кроме того, большинство данных такого класса должны быть защищенными, а не закрытыми. Это порождает их незащищенность от изменений в производных классах и затрудняет тестирование. Там, где с проектной точки зрения нужна большая инкапсуляция, следует применять невиртуальные функции и закрытые данные (§24.3.2.1).

Операции, у которых аргументы не равноправны (один из операндов может трактоваться особо как «объект»), искажают логику проектов, ориентированных исключительно на классовые иерархии. Если же при этом аргументы должны трактоваться равноправным образом, операцию лучше не делать функцией-членом. Это не значит, что она должна стать абсолютно глобальной — ее можно ввести в пространство имен (§24.4).

24.3. Классы

Наиболее фундаментальной идеей объектно-ориентированного проектирования и программирования является взгляд на программу, как на модель определенных аспектов реальности. Классы в программе соответствуют фундаментальным концепциям приложения, и, в частности, фундаментальным концепциям моделируемой реальности. Объекты реальности и артефакты реализации представляются с помощью объектов этих классов.

Анализ взаимоотношений между классами и между разными частями классов — центральная часть проекта системы:

- §24.3.2 Отношения наследования
- §24.3.3 Отношения включения
- §24.3.5 Отношения использования
- §24.3.6 Программируемые отношения
- §24.3.7 Отношения внутри класса

Поскольку класс языка C++ — это тип, классы и их взаимоотношения получают значительную поддержку компилятора и в общем случае поддаются статическому анализу.

Чтобы класс можно было удобным образом вписать в проект, он не только должен представлять собой полезную концепцию, но еще он должен предоставить подходящий интерфейс. Идеальный класс имеет четко определенную минимальную зависимость от остального мира и интерфейс, который выдает минимум информации, необходимой внешнему миру (§24.4.2).

24.3.1. Что представляют собой классы

В системе могут присутствовать два вида классов:

1. Классы, непосредственно представляющие концепции задачи; это концепции, с помощью которых конечные пользователи описывают свои проблемы и их решение.
2. Классы, являющиеся артефактами реализации; это классы, с помощью которых проектировщики и программисты описывают технологии реализации системы.

Некоторые из классов, отражающих артефакты реализации, могут соответствовать и понятиям реального мира. Например, аппаратные и программные ресурсы системы являются хорошими кандидатами в классы приложения. То, что для одного человека — деталь реализации, для другого — характерное понятие конкретной прикладной области.

Хорошо спроектированная система содержит классы, которые поддерживают логически разные взгляды на систему. Например:

1. Классы, непосредственно представляющие пользовательские концепции (например, легковые машины, грузовики и т.д.).
2. Классы, обобщающие пользовательские концепции (например, транспортные средства).
3. Классы, представляющие аппаратные ресурсы (например, память).
4. Классы, представляющие программные ресурсы (например, поток вывода).
5. Классы, помогающие реализовать другие классы (списки, очереди, блокировки).
6. Встроенные типы и управляющие конструкции.

В больших и сложных системах поддерживать четкое разделение между логически разными типами классов и не смешивать уровни абстракции весьма затруднительно. Для простых систем можно ограничиться тремя уровнями абстракции:

1+2. Прикладной взгляд на систему (взгляд со стороны пользователя).

3+4. Аппаратный взгляд на систему (взгляд со стороны компьютера).

5+6. Программный взгляд на систему (взгляд со стороны языка программирования).

Чем больше система, тем большее число уровней абстракции требуется для ее описания, и тем труднее становится разделять эти уровни. Эти уровни абстракции имеют прямые аналогии в природе и человеческой деятельности. Например, можно считать, что дом состоит из:

1. Атомов.
2. Молекул.
3. Бревен и кирпичей.
4. Полов, стен и потолков.
5. Комнат.

Пока эти уровни абстракции поддерживаются раздельно, нет противоречий в вашем взгляде на дом. Но если смешать их произвольным образом, возникают не-

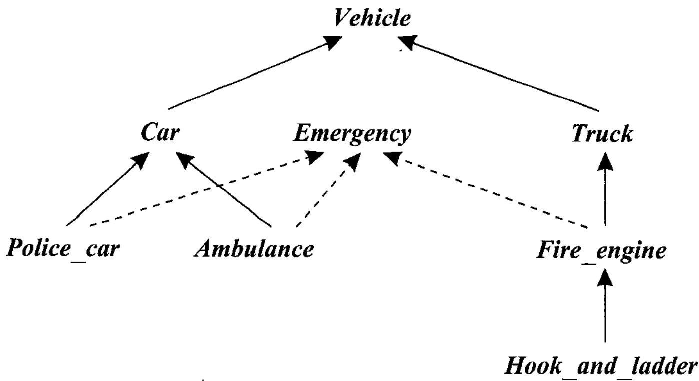
лепости. Например, высказывание «Мой дом состоит из нескольких тысяч фунтов углерода, нескольких видов полимеров, 5000 кирпичей, двух ванных комнат и 13 потолков» звучит глупо. Из-за абстрактной природы компьютерных программ аналогичное по глупости утверждение не так легко распознается.

Переход от концепции прикладной области к классу программного проекта не является простой механической операцией. Тут нужно понимание прикладной области. Отметим, что концепции прикладной области сами являются значительными абстракциями. Например, «налогоплательщики», «монахи» и «сотрудники» не существуют в природе в чистом виде; это просто ярлыки, помогающие классифицировать людей в рамках некоторой системы отсчета (системы социальных понятий). Например, экран монитора не сильно-то напоминает поверхность рабочего стола, хотя он и разработан в виде метафоры рабочего стола, а окна на экране еще меньше напоминают окна в комнате, ветер из которых то и дело сдувает бумаги с реального рабочего стола. Смысл моделирования состоит не в том, чтобы рабски следовать за тем, что мы непосредственно видим и осязаем, а в том, чтобы отталкиваться от этого как от отправной точки в процессе создания проекта, и привязываться как к надежному якорю, когда неосязаемая природа программы грозит опрокинуть наши возможности понимать создаваемый продукт.

Начинающим обычно бывает трудно выявлять классы, но проблема эта рассасывается со временем без долговременных последствий. Затем, однако, наступает фаза, когда классы и классовые иерархии размножаются неконтролируемым образом. Это, как раз, может вызвать долговременные проблемы, усложнив программу и затруднив ее понимание и сопровождение. Не нужно каждую мелкую деталь системы обязательно представлять классом, и не всякую взаимосвязь нужно отражать в форме наследования классов. Цель проектирования — смоделировать систему с соответствующим уровнем детализации и на соответствующем уровне абстракции. Нахождение баланса между простотой и общностью — это не простое дело.

24.3.2. Иерархии классов

Рассмотрим задачу моделирования городского движения, чтобы выявить наиболее вероятное время, необходимое служебным машинам (полиция, скорая помощь, пожарные) для того, чтобы добраться до места назначения. Ясно, что нам потребуется смоделировать машины, грузовики, кареты скорой помощи, пожарные машины разных видов, полицейские машины, автобусы и т.д. Придется применить наследование, ибо перечисленные выше концепции реального мира не существуют изолированно; все они связаны друг с другом множеством связей. Без понимания этих связей нельзя адекватно воспринять концепции. Итак, в программах мы вводим классы для отражения концепций (понятий) и устанавливаем взаимосвязи между ними. Одним из наиболее общих средств представления этих связей является наследование. В нашем примере мы, возможно, захотим особо выделить машины экстренных служб, а также различать легковые машины и грузовые машины. Такой взгляд порождает следующую иерархию:

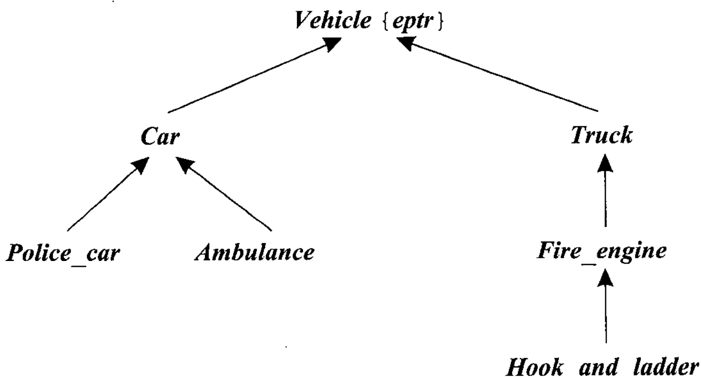


Здесь ***Emergency*** представляет понятие транспортного средства экстренной службы, уместное в нашей задаче моделирования: оно может нарушать обычные правила дорожного движения, имеет преимущества на перекрестках, оно находится на связи с диспетчером и т.д.

Вот соответствующая версия на C++:

```
class Vehicle {/*...*/};
class Emergency {/*...*/};
class Car: public Vehicle {/*...*/};
class Truck: public Vehicle {/*...*/};
class Police_car: public Car, protected Emergency {/*...*/};
class Ambulance: public Car, protected Emergency {/*...*/};
class Fire_engine: public Truck, protected Emergency {/*...*/};
class Hook_and_ladder: public Fire_engine {/*...*/};
```

Наследование — это связь наивысшего уровня, непосредственно представимая языком C++; на ранних стадиях проектирования она также является важнейшей. Часто возникает дилемма — выражать связь посредством наследования, или посредством включения (членства). Например, можно выдвинуть альтернативную концепцию транспортного средства экстренной службы — это транспортное средство, у которого имеется «мигалка». При этом классовая иерархия упрощается, так как класс ***Emergency*** заменяется на соответствующее поле данных в классе ***Vehicle*** (eptr — указатель на «экстренность»):



Теперь класс *Emergency* — это тип полей данных в классах, которым нужно действовать в качестве транспортных средств экстренных служб:

```
class Emergency { /*...*/ };
class Vehicle { protected: Emergency* eptr; /* ... */ };
class Car: public Vehicle { /*...*/ } ;
class Truck: public Vehicle { /*...*/ } ;
class Police_car: public Car { /*...*/ } ;
class Ambulance: public Car { /*...*/ };
class Fire_engine: public Truck { /*...*/ };
class Hook_and_ladder: public Fir_ engine { /*...*/ };
```

Здесь транспортное средство относится к экстренной службе, если указатель *Vehicle::eptr* не равен нулю (соответственно, у простых автомобилей этот член инициализируется нулем). Например:

```
Car::Car()                    // конструктор Car (легковой автомобиль вообще)
   : eptr(0)
{
}
Police_car::Police_car()      // конструктор Police_car (полицейский автомобиль)
       : eptr(new Emergency)
{
}
```

При этом

```
void f(Vehicle* p)
{
  delete p->eptr;
  p->eptr = 0;      // теперь это не транспортное средство экстренной службы

  // ...
  p->eptr = new Emergency;   // снова транспортное средство экстренной службы
}
```

Итак, какой же вариант иерархии классов лучше? Ответ на этот вопрос таков: «тот вариант лучше, что лучше соответствует концепциям реального мира». То есть выбирая между моделями в условиях достижения эффективности и простоты, нужно стремиться к большей реалистичности. В нашем конкретном случае простота преобразования экстренного транспортного средства в обычное кажется мне неестественной. Пожарные машины и кареты скорой помощи имеют специфические конструкции, укомплектованы обученным персоналом и работают в связке с диспетчером, что требует специального коммуникационного оборудования. Отсюда вытекает, что транспортное средство экстренной службы должно моделироваться как фундаментальное понятие, напрямую представимое в программе, что будет способствовать проверке типов и применению автоматизированных инструментов. Если бы мы моделировали другую задачу, где роли автомобилей менее четко определены, где обычные автомобили могли бы перевозить персонал экстренных служб, а устройства связи представлялись простыми сотовыми телефонами, там альтернативная схема моделирования была бы вполне адекватной (достаточно реалистичной).

Для тех, кому пример с моделированием уличного движения кажется надуманным или экзотичным, спешим сообщить, что альтернатива «наследование-включение (членство)» неизменно встречается при проектировании самых разных задач из разных предметных областей. В качестве еще одного примера в §24.3.3 рассматривается полоса прокрутки (scrollbar).

24.3.2.1. Зависимости внутри иерархии классов

Естественно, что производный класс зависит от своего базового класса. Не так часто отмечают и обратную зависимость. Если у класса есть виртуальная функция, значит он полагается на помощь производного класса в реализации части его функциональности. Если же какой-либо член базового класса вызывает одну из виртуальных функций, то в результате базовый класс становится зависимым от производного в своей реализации. Аналогично, если класс использует защищенные члены, то он опять же зависит от производных классов в своей реализации. Рассмотрим пример:

```
class B
{
  // ...
protected:
  int a;

public:
  virtual int f() ;
  int g() {int x =f() ; return x-a; }
};
```

Что делает функция *g*() ? Ответ самым критичным образом зависит от определения *f*() в производных классах. Вот версия, которая гарантирует возврат единицы функцией g():

```
class D1: public B
{
  int f() {return a+1; }
};
```

А вот версия, в которой *g*() выведет "Hello, world!" и вернет назад 0.

```
class D2: public B
{
  int f() {cout<<"Hello, world!\n"; return a; }
};
```

Приведенный пример просто иллюстрирует формальный синтаксис виртуальных функций. Что в этом глупого? Кто-нибудь вполне может написать что-нибудь в этом роде. На самом деле, проблема здесь в том, что виртуальные функции являются частью открытого интерфейса базового класса, который предположительно может использоваться без каких-либо знаний о производных классах. Нужно описать предполагаемое поведение объектов базового класса, чтобы можно было разрабатывать программы, ничего не зная о производных классах. Каждый производный класс, замещающий виртуальную функцию, должен реализовать вариант этого поведения. Например, виртуальная функция *rotate*() класса *Shape* вращает объект этого класса

(фигуру). Функции ***rotate***() из производных классов ***Circle*** и ***Triangle*** должны вращать круги (circles) и треугольники (triangles), соответственно; в противном случае нарушается фундаментальное предположение (представление) о классе ***Shape***. Такого важного предположения о поведении класса В и производных классов ***D1*** и ***D2*** сделано не было; в результате, пример бессмыслен. Даже имена ***B***, ***D1***, ***D2***, ***f*** и ***g*** выбраны так, что понять их предназначение невозможно. Спецификация поведения виртуальных функций — важнейший аспект классового дизайна. Выбор хороших имен для классов и их функций также очень важен (но не всегда легок).

Зависимость от неизвестного (например, еще не написанного) производного класса — это хорошо или плохо? Тут все зависит от намерений программиста. Если нужно изолировать класс от всех внешних влияний так, чтобы можно было гарантировать его специфическое поведение — то виртуальных функций и защищенных членов лучше избегать. Если же целью является построение исходного кодового каркаса, к которому позже другим (или тем же самым) программистом будет добавляться новый код, то виртуальные функции являются элегантным механизмом реализации этой идеи; защищенные члены поддерживают этот механизм. Эта технология положена в основу библиотеки потокового ввода/вывода (§21.6); она также иллюстрируется окончательной версией иерархии ***Ival_box*** (§12.4.2).

Если виртуальная функция предназначена лишь для косвенного использования производным классом, ее можно объявить закрытой. Рассмотрим, например, простой шаблон буфера:

```
template<class T> class Buffer
{
public:
  void put(T);              // вызывает overflow(T) при переполнении буфера
  T get();                  // вызывает underflow(), если буфер пуст
  // ...
private:
  virtual int overflow(T);
  virtual int underflow();
  // ...
};
```

Функции ***put***() и ***get***() вызывают виртуальные функции ***overflow***() и ***underflow***(), соответственно. Пользователь может теперь реализовать разные типы буферов, заместив ***overflow***() и ***underflow***():

```
template<class T> class Circular_buffer: public Buffer<T>
{
  int overflow (T);         // при заполнении возвращаемся на начало буфера
  int underflow ();
  // ...
};

template<class T> class Expanding_buffer: public Buffer<T>
{
  int overflow (T);         // при заполнении увеличиваем размер буфера
  int underflow();
  // ...
};
```

Если бы производному классу нужно было обращаться к ***overflow***() и ***underflow***() напрямую, их следовало сделать защищенными, а не закрытыми.

24.3.3. Агрегация (отношение включения)

Агрегация предполагает использование некоторым классом полей данных, связанных двумя способами с иным классовым типом *X*:

1. Член класса имеет тип *X*.
2. Член класса имеет тип *X** или *X*&.

Если значение указателя никогда не изменяется, эти варианты равносильны (за исключением эффективности и написания конструкторов и деструкторов):

```
class X
{
public:
  X(int);
  //...
};

class C
{
  X a;
  X* p;
  X& r;

public:
  C(int i, int j, int k) : a(i), p(new X(j)), r(*new X(k)) {}
  ~C() {delete p; delete &r;}
};
```

В таких случаях включение самого объекта, то есть *C*::*a* в данном примере, эффективнее по быстродействию и памяти (и даже по нажатиям клавиш). Это и надежнее, ибо связь между внешним (содержащим) и внутренним (содержащимся) объектами определяется правилами конструирования и уничтожения (§10.4.1, §12.2.2, §14.4.1). См. также §24.4.2 и §25.7.

Указатель следует применять тогда, когда за время жизни «содержащего» объекта требуется изменить значение указателя на «содержащийся» объект. Например:

```
class C2
{
  X*p;

public:
  C2(int i) : p(new X(i)) {}
  ~C2() {delete p;}

  X* change(X* q)
  {
    X* t = p;
    p = q;
    return t;
  }
};
```

Другое соображение в пользу применения указателя состоит в том, чтобы разрешить задавать «содержащийся» объект в качестве аргумента:

```
class C3
{
  X*p;

public:
  C3(X* q) : p(q) { }
  // ...
};
```

Создавая объекты, содержащие указатели на другие объекты, мы получаем то, что часто называют *иерархией объектов* (*object hierarchies*). Это альтернатива иерархиям классов. Как показано в примере с транспортными средствами экстренных служб в §24.3.2, в процессе проектирования нелегко выбрать способ отражения свойства класса — в виде базового класса, или в виде содержащегося в классе члена. Первое предпочтительнее в случае, когда требуется замещение. Второе же предпочтительнее тогда, когда свойство может соответствовать разным типам. Например:

```
class XX: public X { /* ... */ };
class XXX: public X { /* ... */ };

void f()
{
  C3* p1 = new C3(new X);      // C3 "содержит" X
  C3* p2 = new C3(new XX);     // C3 "содержит" XX
  C3* p3 = new C3(new XXX);    // C3 "содержит" XXX
  // ...
}
```

Это нельзя смоделировать наследованием *C3* от *X* или включением в *C3* члена типа *X*, ибо точный тип члена становится известен позже. Это также важно для классов с виртуальными функциями, например, для класса фигур (§2.6.2) или абстрактного класса множеств (§25.3).

Вместо классов, содержащих указатели, можно для упрощения применить класс, содержащий ссылку, если только требуется работать с единственным включенным объектом. Например:

```
class C4
{
  X& r;

public:
  C4(X& q) : r(q) { }
  // ...
};
```

Члены-указатели и члены-ссылки также нужны для совместного использования объекта:

```
X* p = new XX;
C4 obj1(*p);
C4 obj2(*p);                   // obj1 и obj2 разделяют новый XX
```

Естественно, что управление совместно используемым объектом требует повышенного внимания — особенно в параллельных системах.

24.3.4. Агрегация и наследование

С учетом важности отношений наследования не покажется удивительным, что ими часто злоупотребляют. Когда класс ***D*** открыто наследует классу ***B***, говорят, что ***D*** *есть* ***B*** (***D*** *is a* ***B***):

```
class B {/* ...*/};
class D: public B {/*...*/}; // D - это разновидность B
```

Говорят даже, что наследование есть *отношение "is-a"* (*is-a relationship*) или несколько точнее, что ***D*** *есть разновидность* ***B*** (***D is a kind of B***). Если же класс ***D*** содержит член типа ***B***, говорят, что ***D*** *агрегирует* (содержит — *contains*) B. Например:

```
class D                    // D содержит (агрегирует) B
{
public:
  B b;
  //...
};
```

Про этот тип взаимосвязи говорят, что это *отношение типа "иметь" (has-a relationship)*.

Для заданных классов ***B*** и ***D***, как выбрать между наследованием и агрегацией? Рассмотрим для примера классы ***Airplane*** (самолет) и ***Engine*** (двигатель). Новичкам часто приходит в голову сделать ***Airplane*** производным от ***Engine***. Это плохая идея, поскольку самолет не является двигателем, он содержит (агрегирует, имеет) двигатель. Подумайте, может ли самолет иметь несколько двигателей? Поскольку это возможно, нам следует применить агрегирование, а не наследование. Во многих других случаях стоит задать вопрос — может ли быть несколько экземпляров одной сущности у другой сущности? Неосязаемая природа программ заставляет нас задавать такие вопросы. Если бы все классы были столь наглядными, как классы ***Airplane*** и ***Engine***, было бы легко избежать таких тривиальных ошибок, когда ***Airplane*** делают производным от ***Engine***. Однако в реальной практике подобные ошибки случаются весьма часто, особенно у людей, считающих наследование просто механизмом комбинирования свойств на уровне конструкций языка программирования. Несмотря на определенные преимущества наследования в виде краткости и удобства записи, его следует применять исключительно для выражения идеи общности/подтипизации, отраженной в проектных спецификациях. Рассмотрим пример:

```
class B
{
public:
  virtual void f();
  void g();
};

class D1              // D1 содержит B
{
```

```
public:
  B b;
  void f();           // не замещение b.f()
};

void h1(D1* pd)
{
  B* pb = pd;         // error: нет преобразования от D1* к B*
  pb = &pd->b;
  pb->g();            // вызывается B::g()
  pd->g ();           // error: у D1 нет члена-функции g()
  pd->b.g();
  pb->f();            // вызывается B::f()
  pd->f();            // вызывается D1::f()
}
```

Заметьте, что не существует неявных преобразований от типа класса, содержащего член некоторого другого класса, к этому другому классу, и что первый из перечисленных классов не замещает виртуальных функций второго. Это резко контрастирует со случаем открытого наследования:

```
class D2: public B
{
public:
  void f();           // замещает B::f()
};

void h2(D2* pd)
{
  B* pb = pd;         // ok: неявное приведение от D2* к B*
  pb->g();            // вызывается B::g()
  pd->g();            // вызывается B::g()
  pb->f();            // виртуальный вызов: вызывается D2::f()
  pd->f();            // вызывается D2::f()
}
```

Удобство и большая краткость записи в случае класса ***D2*** настолько превосходит таковое в случае класса ***D1***, что этим легко злоупотребить. Стоит помнить, что за удобство записи приходится платить большей степенью зависимости между ***B*** и ***D2*** (см. §24.3.2.1). В частности, стоит помнить о неявном преобразовании из ***D2*** в ***B***. Если такое преобразование не является приемлемой частью семантики ваших классов, открытого наследования стоит избегать. Когда же класс отражает некоторую сущность, а наследование от него используется в проекте для отражения зависимостей типа "is-a", то указанное неявное преобразование часто оказывается именно тем, что нужно.

Наконец, бывают случаи, когда наследование вполне подходит, но неявного преобразования нужно избежать. Рассмотрим класс ***Cfield***, который помимо прочего обеспечивает динамическое (во время выполнения программы) управление доступом для другого класса ***Field***. На первый взгляд, сделать ***Cfield*** производным от ***Field*** вполне разумно:

```
class Cfield: public Field {/* ... */};
```

Таким образом, с помощью наследования выражается тот факт, что ***Cfield*** является разновидностью ***Field***, повышается удобство записи при использовании функциями ***Cfield*** общих с ***Field*** членов, и — самое важное — позволяет классу ***Cfield*** замещать виртуальные функции класса ***Field***. Подвох, однако, заключается в том, что неявное преобразование от ***Cfield**** к ***Field****, вытекающее из определения типа ***Cfield***, сводит на нет все попытки контролировать доступ к ***Field***:

```
void g (Cfield* p)
{
  *p = "asdf" ;   // доступ к Field при контроле со стороны операции присв-ния из Cfield:
                  // p->Cfield::operator=("asdf")

  Field* q = p;   // неявное приведение от Cfield* к Field*
  *q = "asdf" ;   // OOPS! нет контроля
}
```

Другое решение — сделать так, чтобы ***Cfield*** агрегировал (содержал) член типа ***Field***, но это не позволит ***Cfield*** замещать виртуальные функции класса ***Field***. Наилучшим решением в таком случае будет *закрытое наследование* (*private derivation*):

```
class Cfield: private Field { /* ... */ };
```

С проектной точки зрения закрытое наследование эквивалентно агрегации (включению) за существенным исключением касательно замещения. Важный пример применения такого подхода — открытое наследование от абстрактного базового класса, определяющего интерфейс, и одновременно закрытое или защищенное наследование от конкретного класса с целью предоставления реализации (§2.5.4, §12.3, §25.3). Поскольку закрытое или защищенное наследование является деталью реализации, не оказывающей влияния на тип производного класса, его иногда называют *наследованием реализации* (*implementation inheritance*), что контрастирует с открытым наследованием, подразумевающим наследование интерфейса и неявное преобразование в базовый тип. Это часто называют *подтипизацией* (*subtyping*) или *наследованием интерфейса* (*interface inheritance*).

Другим способом выразить то же самое является утверждение, что «объект производного класса можно использовать там, где допустим объект базового класса». Это часто называют *«принципом подстановки Liskov»* (§23.6[Liskov,1987]) (*Liskov substitution principle*). Различие между public/protected/private формами наследованиями ярко наблюдается для полиморфных типов, объекты которых адресуются через указатели или ссылки.

24.3.4.1. Альтернатива «включение/наследование»

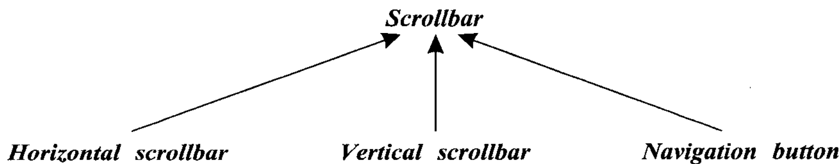
Дальнейшее изучение альтернативы «включение (агрегация)/наследование» будем проводить на примере графического интерфейса пользователя: зададимся вопросом, как представить полосу прокрутки (scrollbar) в интерактивной графической системе и как прикрепить ее к окну. Нам нужны два вида полос прокрутки — горизонтальная и вертикальная. Мы можем представить их двумя типами — ***Horizontal_scrollbar*** и ***Vertical_scrollbar*** — или одним типом ***Scrollbar***, принимающим аргумент, определяющий пространственное расположение (горизонтальное или вертикальное). В первом случае нужен еще и третий тип — ***Scrollbar*** (полоса прокрутки вообще), выступающий в качестве базового класса для двух частных ти-

пов полос прокрутки. Во втором случае нужен дополнительный аргумент для задания типа полосы прокрутки и два его значения для идентификации этих вариантов полос. Например:

```
enum Orientation {horizontal, vertical};
```

Выбрав один из двух вариантов, мы сталкиваемся с необходимостью дальнейших расширений системы. Например, может обнаружиться необходимость ввести третий тип полосы прокрутки. Первоначально мы могли думать, что существует лишь две возможности для полос прокрутки («в конце концов, окно имеет лишь два измерения»). И, тем не менее, есть и другие возможности, требующие изменений в первоначальном проекте. Например, кому-то может прийти в голову использовать одну «навигационную кнопку» вместо двух полос прокрутки. Такая кнопка может осуществлять прокрутку в разных направлениях в зависимости от того, где пользователь нажал эту кнопку: если где-то вверху посередине — то прокрутка выполняется вертикально вверх; если слева — то прокрутка выполняется горизонтально влево; если в районе верхнего левого угла — то одновременная прокрутка горизонтально влево и вертикально вверх и т.д. Такие кнопки встречаются в реальной практике. Их можно рассматривать как развитие идей полос прокрутки.

Добавление навигационной кнопки в программу с иерархией трех классов полос прокрутки означает добавление нового класса, но это не влечет за собой никаких изменений в коде старых полос прокрутки:



Это очень приятный аспект иерархического решения.

Задание ориентации полосы прокрутки в качестве аргумента означает наличие полей типа в объектах полос прокрутки и наличие операторов ***switch*** в коде функций-членов. Таким образом, проектный выбор между двумя вариантами системы полос прокрутки означает выбор между объявлениями и лишним кодом. Первый вариант увеличивает степень статической проверки и увеличивает объем информации, доступной автоматизированным инструментам. Второй вариант откладывает выбор до момента исполнения программы и позволяет вносить изменения посредством модификации кода индивидуальных функций, не касаясь общей структуры системы, как она видится с точки зрения проверки типов и иных инструментальных средств. Лично я предпочитаю классовую иерархию, непосредственно моделирующую иерархические отношения между концепциями системы.

Решение с одним типом полосы прокрутки облегчает задачу хранения и передачи информации о типе полосы:

```
void helper (Orientation oo)
{
  // ...
  p = new Scrollbar (oo) ;
```

```
  // ...
}

void me ()
{
  helper (horizontal) ;
  // ...
}
```

Такое решение может облегчить переориентацию полос прокрутки прямо во время выполнения программы. Вряд ли здесь это имеет большое значение, но во многих иных случаях это важно.

В общем, альтернатива есть, но выбор не прост.

24.3.4.2. Альтернатива «агрегация/наследование»

Теперь рассмотрим вопрос о том, как привязать полосу прокрутки к окну. Если мы будем считать класс ***Window_with_scrollbar*** чем-то таким, что одновременно является и окном, и полосой прокрутки, то получим нечто вроде

```
class Window_with_scrollbar : public Window, public Scrollba
{
  // ...
};
```

Это позволяет любому ***Window_with_scrollbar*** действовать и как ***Scrollbar***, и как ***Window***, но ограничивает нас одним типом полосы прокрутки.

С другой стороны, если мы будем рассматривать ***Window_with_scrollbar*** как окно, содержащее полосу прокрутки, то получим следующее:

```
class Window_with_scrollbar : public Window
{
  // ...
  Scrollbar* sb;

public:
  Window_with_scrollbar (Scrollbar* p, /* ... */) :
                        Window (/* ...*/), sb (p) {/*...*/}
  // ...
};
```

Это позволяет применить решение с иерархией полос прокрутки. Передача типа полосы в качестве аргумента позволяет окну не знать точного типа своей полосы прокрутки. Мы даже можем передать ***Scrollbar*** так же, как мы передавали ориентацию ***Orientation*** (§24.3.4.1). Если понадобится, чтобы ***Window_with_scrollbar*** действовал как полоса прокрутки, достаточно добавить операцию преобразования:

```
Window_with_scrollbar : : operator Scrollbar& ()
{
  return *sb;
}
```

На мой взгляд, предпочтительнее, чтобы окно содержало полосу прокрутки. Мне легче представить окно, имеющее полосу прокрутки, чем окно, которое по совместительству еще и полоса прокрутки. Моя излюбленная стратегия проектирования со-

стоит в том, что полоса прокрутки есть специальный вид окон, который содержится в окнах, нуждающихся в полосах прокрутки. Такая стратегия отдает предпочтение агрегации перед наследованием. Другой аргумент в пользу этого решения проистекает из эмпирического правила, заключающегося в вопросе «может ли окно содержать несколько полос прокрутки» (§24.3.4). Поскольку это действительно так (почему окно не может содержать, например, горизонтальную и вертикальную полосы прокрутки), класс ***Window_with_scrollbar*** не следует делать производным от ***Scrollbar***.

Заметьте, что невозможно создать производный класс от неизвестного класса. Точный тип базового класса должен быть известен на этапе компиляции (§12.2). С другой стороны, если атрибут класса передается конструктору в качестве параметра, в классе где-нибудь должно быть представляющее его поле данных. Если тип этого поля (члена класса) — указатель или ссылка, мы можем передать объект класса, производного от типа, ассоциированного с указателем или ссылкой. В частности, поле данных ***sb*** типа ***Scrollbar**** из предыдущего примера может реально указывать на объекты полос прокрутки типа ***Navigation_button***, неизвестные пользователям ***Scrollbar****.

24.3.5. Отношение использования

Для проекта важно знать и понимать, как класс использует другие классы. Такие зависимости проявляются в языке C++ неявно. Класс может использовать только имена, которые были где-то объявлены. Для извлечения всех используемых имен требуется либо специальный инструмент, либо внимательное чтение файлов проекта. Способы, которыми класс *X* использует класс *Y*, можно классифицировать по-разному, например следующим образом:

- *X* использует имя *Y*.
- *X* использует *Y*.
  - *X* вызывает функцию-член *Y*.
  - *X* читает поле данных класса *Y*.
  - *X* записывает поле данных класса *Y*.
- *X* создает объект типа *Y*.
  - *X* выделяет память под *автоматические* или *статические* объекты *Y*.
  - *X* создает объект класса *Y* при помощи ***new***.
- *X* принимает размеры *Y*.

Получение в качестве аргумента размеров объекта классифицируется отдельно, поскольку в этом случае требуется знакомство с объявлением класса, но нет зависимости от конструкторов. Пользование именем *Y* также классифицируется отдельно, поскольку используя *Y** или упоминая *Y* в объявлении внешней функции, мы не нуждается в знании объявления класса *Y* (§5.7):

```
class Y;        // Y - имя класса
Y* p;
extern Y f(const Y&);
```

Часто важно различать зависимости от интерфейса класса (объявления класса) и зависимости от реализации класса (определение классовых функций-членов).

Последние имеют еще много других зависимостей, менее интересных пользователю, чем зависимость от интерфейса класса (§24.4.2). Как правило, целью проектирования является минимизация зависимостей от интерфейса, ибо они становятся зависимостями пользователей класса (§8.2.4.1, §9.3.2, §12.4.1.1, §24.4).

Язык C++ не требует от программиста специфицировать, какие другие классы используются и каким образом. Отчасти это связано с тем, что число таких зависимостей может быть весьма велико, что при полном их описании чрезвычайно затруднит читабельность кода. И, вообще, это не вопрос языка программирования. Скорее, это вопрос проектирования системы, который может поддерживаться специальными инструментами.

24.3.6. Программируемые отношения

Язык программирования не может и не должен напрямую поддерживать любую и каждую концепцию проектирования. Подобным образом и методы, и технологии проектирования не должны поддерживать все особенности любого языка программирования. Язык проектирования обычно богаче и менее озабочен деталями, чем язык программирования. Язык программирования должен быть способен поддержать множество идей (идиом) проектирования, или он потерпит неудачу в приспособлении к проекту.

Когда язык программирования не может напрямую поддержать некоторую идею проектирования, нужно косвенно отобразить проектную конструкцию в конструкции языка программирования. Пусть, например, в проекте введено понятие *делегирования* (*delegation*) посредством требования, чтобы все операции, не определенные в классе *A*, передавались (делегировались) для выполнения объекту класса *B*, адресуемому указателем *p*. Язык C++ не поддерживает идиому делегирования напрямую, но ее можно смоделировать в коде на C++ (то есть запрограммировать) следующим образом:

```
class B
{
  // ...
  void f() ;
  void g() ;
  void h() ;
};

class A
{
  B* p;                       // делегируем через указатель p
  // ...
  void f() ;
  void ff() ;
  void g() {p->g() ; }        // делегируем g()
  void h() {p->h() ; }        // делегируем h()
};
```

Программисту ясно, что здесь класс *A* делегирует работу классу *B* через указатель *A::p*, но для языка программирования нет никакой разницы, как именно используется указатель типа *B**. Соответственно, такие «программируемые отношения» между классами плохо распознаются автоматизированными инструментами.

Взаимооднозначные соответствия между концепциями проекта и средствами языка программирования следует использовать всегда, когда только возможно. Они обеспечивают простоту и гарантируют, что проект точно отражен в коде, а это помогает в работе и программистам, и автоматизированным инструментам.

Операции преобразования обеспечивают языковый механизм для выражения программируемых отношений. Например, операция преобразования *X*: : ***operator*** ***Y***( ) утверждает, что если допустим *Y*, то можно использовать и *X* (§11.4.1). Конструктор ***Y***: : ***Y***(***X***) выражает то же самое отношение. Отметим, что операция преобразования (и конструктор) производят новый объект, а не изменяют тип существующего. Объявляя операцию преобразования к *Y*, мы обеспечиваем способ неявного вызова функции, возвращающей *Y*. Поскольку неявное применение преобразований, программируемых с помощью соответствующих операций и конструкторов, может быть опасным, их следует тщательно и по отдельности проанализировать в проекте.

Важно также гарантировать, что графы преобразования типов не зацикливаются. В противном случае возникнут ошибки для некоторых комбинаций типов. Например:

```
class Rational;

class Big_int
{
public:
  friend Big_int operator+ (Big_int, Big_int) ;
  operator Rational () ;
  // ...
};

class Rational
{
public:
  friend Rational operator+ (Rational, Rational) ;
  operator Big_int () ;
  // ...
};
```

Типы ***Rational*** и ***Big_int*** взаимодействуют не так гладко, как кто-то надеялся:

```
void f(Rational r, Big_int i)
{
  g (r+i) ;  // error, неоднозначность: operator+(r,Rational(i)) или operator+(Big_int(r),i)?

  g (r+Rational (i) ) ;  // одно явное разрешение
  g (Big_int (r) +i) ;   // другое явное разрешение
}
```

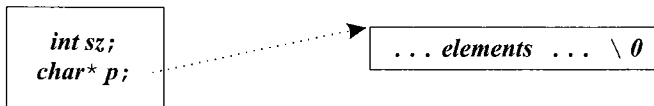
Можно избежать таких взаимных преобразований, сделав некоторые из них явными. Например, преобразование ***Big_int в Rational*** можно определить (вместо операции) функцией ***make_Rational***( ), и тогда сложение разрешилось бы как ***g***(***Big_int***(***r***) , ***i***). Там, где не избежать взаимных операций преобразования, следует разрешать конфликты при помощи явных преобразований или предоставляя несколько версий таких операций, как +.

24.3.7. Отношения внутри класса

Класс может скрывать любое число деталей реализации и некоторое количество «грязи». Иногда ему это приходится делать. В то же время, объекты класса имеют вполне регулярную структуру и ими можно легко манипулировать. Объект класса — это набор некоторых подобъектов (часто называемых членами объекта), многие из которых являются указателями или ссылками на другие объекты. Таким образом, объект можно представить себе в виде корня дерева других объектов, в совокупности составляющих иерархию объектов, дополнительную к иерархии классов (§24.3.2.1). Например, рассмотрим очень простой класс ***String***:

```
class String
{
  int sz;
  char* p;
public:
  String(const char* q);
  ~String();
  //...
};
```

Объект класса ***String*** можно представить графически следующим образом:



24.3.7.1. Инварианты

Значения членов объекта и значения объектов, на которые могут указывать члены, в совокупности называются *состоянием объекта* или его *значением* (*object state* or *value*). Главной заботой при проектировании класса является необходимость привести объект в точно определенное состояние (инициализация/конструирование), поддерживать такое состояние в процессе выполнения операций с объектом и, наконец, уничтожить объект корректным и изящным образом. Свойство, делающее состояние объекта четко определенным, называется *инвариантом* (*invariant*).

Итак, целью проектирования является приведение объекта в состояние с выполняющимся инвариантом. Как правило, это делается конструктором. Любая операция над классом ожидает истинный инвариант на входе и обязана оставить инвариант в этом состоянии на выходе. В конце концов, деструктор отменяет инвариант, уничтожая объект. Например, конструктор ***String::String(const char*)*** гарантирует, что ***p*** указывает на массив по крайней мере из ***sz+1*** элементов, где ***sz*** имеет разумное значение и ***p[sz]==0***. Каждая строковая операция должна оставлять это утверждение истинным.

Искусство проектирования состоит в том, чтобы сделать класс простым и чтобы его реализация имела четко выраженный инвариант. Заявить это просто, но не так-то просто реально сформулировать полезный инвариант, который бы легко воспринимался и не налагал неприемлемых ограничений на эффективность операций класса. Под инвариантом мы будем здесь понимать фрагмент кода, который

можно выполнить для проверки состояния объекта. Можно также использовать и более строгое, математическое определение инварианта, и оно бывает уместно в некоторых случаях. Но мы будем придерживаться более простого понятия инварианта, как практической (пусть и логически неполной) проверки состояния объекта.

Понятие инварианта ведет свое начало от работ Флойда, Наура и Хоара (Floyd, Naur, Hoare) о предусловиях и постусловиях и присутствует практически во всех работах по абстрактным типам данных и верификации программ за последние 30 лет. Оно также широко применяется в процессе отладки программ, написанных на языке C.

Как правило, инвариант не поддерживается в процессе выполнения функций-членов. Функции, которые могут вызываться в этот момент, не должны быть частью открытого интерфейса класса (здесь лучше применять защищенные и закрытые функции).

Как выразить понятие инварианта в коде на C++? Простейший путь — определить функцию проверки инварианта и вставлять ее вызов в открытые операции. Например:

```
class String
{
  int sz;
  char* p;

public:
  class Range { };                          // классы исключений
  class Invariant { };

  enum { TOO_LARGE = 16000 };               // предел длины

  void check ();                            // проверка инварианта

  String (const char* q);
  String (const String&);
  ~String ();

  char& operator[] (int i);
  int size () {return sz; }
  //...
};

void String::check ()
{
  if(p==0 || sz<0 || TOO_LARGE<=sz || p[sz]) throw Invariant ();
}

char& String::operator[] (int i)
{
  check ();                                 // проверка на входе
  if(i<0 || sz<=i) throw Range ();          // работаем
  check ();                                 // проверка на выходе
  return p[i];
}
```

Это хорошо работает и не представляет какой-либо сложности для программиста. В то же время, для такого простого класса, как ***String***, проверки займут боль-

шую часть времени работы методов и, возможно, большую часть объема кода. Поэтому программисты часто выполняют проверку инварианта только во время отладки:

```
inline void String::check()
{
#ifndef NDEBUG
  if(p==0||sz<0||TOO_LARGE<=sz||p[sz]) throw Invariant();
#endif
}
```

Здесь макрос *NDEBUG* используется так же, как в стандартном макросе *assert*() языка С. Принято устанавливать *NDEBUG* в знак того, что отладка не производится.

Столь простая работа с инвариантами во время отладки оказывает бесценную помощь в получении правильно работающего кода и, что даже еще важнее, в четком отражении классами концепций проекта. Дело в том, что при определении инвариантов приходится рассматривать классы с иной точки зрения, в результате чего код получает полезную избыточность, увеличивая вероятность выявления ошибок, несовместимостей и оплошностей.

24.3.7.2. Утверждения

Инвариант — это особая форма *утверждения* (*assertion*). Утверждение же в общем случае есть просто высказывание о том, что некоторое логическое условие выполняется. Вопрос здесь только один — что делать, если условие не выполняется.

Стандартная библиотека языка С — и стало быть С++ — предоставляет макрос *assert*() в заголовочном файле <*assert.h*> или <*cassert*>. Макрос *assert*() вычисляет значение аргумента и вызывает *abort*() в случае нулевого результата (*false*). Например:

```
void f(int* p)
{
  assert(p!=0);        // утверждается, что p!=0; abort() если p равно нулю
  // ...
}
```

Перед тем, как прекратить работу программы, *assert*() выводит имя своего исходного файла и номер строки, на которой это произошло. Это делает макрос *assert*() полезным отладочным средством. Обычно *NDEBUG* устанавливается с помощью опций компилятора отдельно для каждой единицы трансляции. Отсюда следует, что не стоит использовать *assert*() во встраиваемых и шаблонных функциях, поскольку их определения включаются в разные единицы трансляции, и нужно тщательно следить, чтобы *NDEBUG* был всюду установлен одинаково (§9.2.3). Как и все, что относится к магии макросов, применение *NDEBUG* следует признать весьма низкоуровневым приемом, довольно запутанным и подверженным ошибкам. Кроме того, иногда нужно оставлять диагностические средства и в хорошо работающей программе, а вызов *abort*() не подходит для готового кода промышленного уровня.

Альтернативой является применение шаблона *Assert*(), который генерирует прерывание, а не безусловное прекращение работы программы, так что его можно оста-

вить в окончательном варианте кода (если это необходимо). К сожалению, ***Assert***() не входит в состав стандартной библиотеки, но он очень просто определяется:

```
template<class X, class A> inline void Assert(A assertion)
{
  if(!assertion) throw X();
}
```

Шаблон ***Assert***() генерирует исключение ***X***(), если условие ***assertion*** ложно. Например:

```
class Bad_arg { };

void f(int* p)
{
  Assert<Bad_arg> (p!=0);
  // ...
}
```

Так как здесь условия утверждений формулируются явно, то можно явным образом указать необходимость проверки, например, только в условиях отладки:

```
void f2(int* p)
{
  Assert<Bad_arg> (NDEBUG || p!=0);
  // ...
}
```

Применение здесь операции `||` вместо `&&` может показаться удивительным. Тем не менее, ***Assert<E>***(***a***||***b***) проверяет истинность !(***a***||***b***), что эквивалентно !***a*** && !***b***.

Указанное применение ***NDEBUG*** требует установки для него соответствующего значения в зависимости от того, осуществляем мы отладку, или нет. Поскольку реализации C++ не делают этого по умолчанию, лучше самим установить некоторое характерное значение. Например:

```
#ifdef NDEBUG
const bool ARG_CHECK = false;
#else
const bool ARG_CHECK = true;
#endif

void f3(int* p)
{
  Assert<Bad_arg> (!ARG_CHECK || p!=0);
  // ...
}
```

Если генерируемое в процессе проверки утверждения прерывание не перехватывается, то ***Assert***() заканчивает выполнение программы вызовом ***terminate***() (почти так же, как ***assert***() заканчивает программу вызовом ***abort***()). Обработчик прерывания может выполнить и другие, менее радикальные действия.

Обычно, при тестировании любой программы реальных (не учебных) размеров я включаю или выключаю проверочные утверждения группами. Применение ***NDEBUG*** — это простейший способ реализации такой техники. На ранних стадиях раз-

работки включается большинство проверочных утверждений, в то время как в окончательном коде их остается совсем немного (лишь критически важные проверки). Как видно из приведенного примера, такой стиль управления отладкой поддерживается разбиением утверждений на две части — одна часть является условием, которое управляет включением/выключением проверки (например, *ARG_CHECK*), а вторая содержит собственно проверяемое условие.

Если включающее/выключающее условие является константным выражением, то при его запрещающем значении все утверждение будет удалено компилятором. Однако включающее/выключающее условие может быть и переменной, значение которой можно динамически изменять во время выполнения программы в соответствии с нуждами отладки. Например:

```
bool string_check = true;

inline void String::check()
{
  Assert<Invariant>(!string_check || (p && 0<=sz && sz<TOO_LARGE && p[sz]==0));
}

void f()
{
  String s = "wonder";
  // здесь строки проверяются
  string_check = false;
  // а здесь строки не проверяются
}
```

Естественно, в этом случае генерируется реальный код, так что стоит следить за тем, чтобы его проверочная часть не разбухла слишком сильно из-за интенсивного применения утверждений.

Отметим, что оператор

```
Assert<E>(a);
```

эквивалентен

```
if(!a) throw E();
```

В связи с этим возникает вопрос, зачем связываться с *Assert*(), а не просто написать этот оператор? Дело в том, что применение *Assert*() четко отражает намерение проектировщика. Оно явным образом говорит, что некоторое условие предполагается в этом месте программы истинным. Это не обычный программный код, а ценная дополнительная информация для любого человека, читающего программу. Еще одним (более приземленным) преимуществом является большая простота обнаружения *assert*() или *Assert*() в большом коде, нежели нахождение в нем условных выражений, генерирующих исключения.

Шаблон *Assert*() может быть обобщен так, чтобы он генерировал переменные исключения и исключения с аргументами:

```
template<class A, class E> inline void Assert(A assertion, E except)
{
  if(!assertion) throw except;
}
```

```
struct Bad_g_arg
{
  int* p;
  Bad_g_arg(int* pp) : p(pp) { }
};

bool g check = true;
int g max = 100;

void g(int* p, exception e)
{
  Assert(!g_check || p!=0, e);
  Assert(!g_check || (0<*p&&*p<=g_max), Bad_g_arg(p));
  //...
}
```

К сожалению, многие компиляторы не сумеют устранить избыточный код для обобщенного *Assert*() в случаях, когда условие вычисляется на этапе компиляции. Поэтому двухаргументные *Assert*() следует применять лишь в случае, когда исключение не представимо в форме *E*(), или когда код нужно генерировать независимо от значения утверждения.

В §23.4.3.5 рассматривались две наиболее распространенные формы реорганизации классовых иерархий — расщепление классов на два класса и выделение общей части двух классов в общий базовый класс. В обоих случаях большую помощь могут оказать грамотно спроектированные инварианты. В классе, созревшем для расщепления, хорошо видна избыточность большей части проверок инварианта, сосредоточенных в его операциях; после расщепления подмножества операций будут иметь дело с соответствующими подмножествами состояний объекта. В классах с общими частями будут просматриваться схожие инварианты, несмотря на различие в деталях реализации.

24.3.7.3. Предусловия и постусловия

Популярным применением утверждений является формулировка предусловий и постусловий для функций. Имеется ввиду проверка основных предположений относительно входа в функцию и проверка того, что при выходе функция оставляет мир в надлежащем состоянии. К сожалению, часто подобные утверждения формулируются на более высоком уровне, так что язык программирования не всегда в состоянии отразить их удобным образом. Например:

```
template<class Ran> void sort(Ran first, Ran last)
{
  Assert<Bad_sequence>("[first,last) is a valid sequence");    // псевдокод

  // ... сортирующий алгоритм ...

  Assert<Failed_sort>("[first,last) is in increasing order");  // псевдокод
}
```

Это фундаментальная проблема. Общие высказывания о программе лучше формулируются на высокоуровневом языке, близком к математическому, а не на алгоритмическом языке, на котором пишутся программы.

В связи с инвариантами требуется немалая доля интеллекта, чтобы перевести высказываемое нами идеализированное утверждение в нечто такое, что можно проверить алгоритмически. Например:

```
template<class Ran> void sort(Ran first, Ran last)
{
  // [first,last) - допустимая последовательность:
  Assert<Bad_sequence>(NDEBUG || first<=last);

  // ... сортирующий алгоритм ...

  // [first,last) находится в возрастающем порядке:
  Assert<Failed_sort>( NDEBUG ||
                      (last-first<2 || (*first<=last[-1] &&
                      *first<=first[(last-first)/2] &&
                      first[(last-first)/2]<=last[-1])));
}
```

Несмотря на то, что писать обычные проверки аргументов и результатов вычислений намного проще, чем утверждения, я все же рекомендую обязательно попытаться выразить идеальные предусловия и постусловия (хотя бы в виде комментариев), прежде чем свести их к чему-либо менее абстрактному, что можно легко выразить на языке программирования.

Проверка предусловий может легко выродиться в тривиальную проверку значений аргументов вызова. Поскольку аргументы часто передаются многим функциям, такая проверка становится дорогой. Кроме того, проверка, например, всех аргументов-указателей на неравенство нулю может вызвать необоснованное успокоение и дать ложное чувство уверенности, особенно если оно выполняется лишь в отладочных вариантах программы. Вот почему я рекомендую сосредоточить внимание на инвариантах.

24.3.7.4. Инкапсуляция

Заметьте, что в языке C++ именно класс, а не объект класса, является единицей инкапсуляции. Например:

```
class List
{
  List* next;
public:
  bool on(List*);
  // ...
};

bool List::on(List* p)
{
  if(p == 0) return false;
  for(List* q = this; q; q=q->next) if(p==q) return true;
  return false;
}
```

Доступ к закрытому указателю ***List::next*** разрешен, ибо функция ***List::on()*** имеет доступ к любым объектам класса ***List***, так или иначе попадающим в зону ее действия. Там, где это неудобно, можно упростить ситуацию и без эксплуатации возможностей функций-членов иметь открытый доступ к объектам того же самого класса. Например:

```
bool List::on(List* p)
{
  if(p == 0) return false;
```

```
    if(p == this) return true;
    if(next == 0) return false;
    return next->on(p);
}
```

Здесь, однако, итерация превращается в рекурсию, а это может нанести удар по быстродействию в случае, если компилятор не способен на оптимизацию, превращающую рекурсию обратно в итерацию.

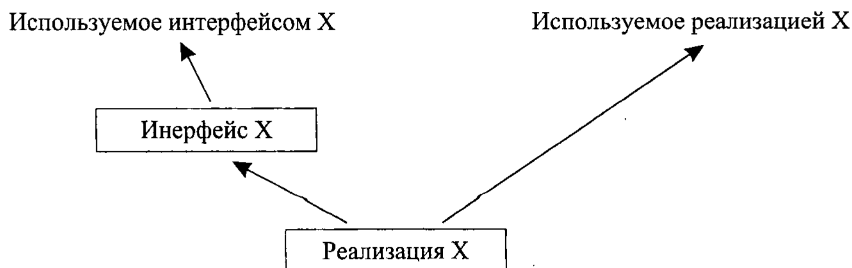
24.4. Компоненты

Единицей проектирования является набор классов, функций и т.д., а не отдельный класс. Это множество часто также называют библиотекой или каркасом (§25.8), и оно является предметом повторного использования (§23.5.1), сопровождения и т.д. Язык C++ предоставляет три возможности для выражения совокупности средств, объединенных неким единым логическим критерием:

1. Класс — коллекция данных, функций, шаблонов и типизированных членов.
2. Иерархия классов — коллекция классов.
3. Пространство имен — коллекция данных, функций, шаблонов и типизированных членов.

Класс обеспечивает средства, делающие удобным создание объектов типа, представляемого этим классом. В то же время, многие важные компоненты не могут быть описаны механизмом, удобным для создания объектов единственного типа. Иерархия классов предназначена для отражения понятия множества родственных классов. Однако составные части компонентов не всегда представимы именно классами, и не все классы можно вписать в осмысленные иерархии (§24.2.5). Поэтому самым непосредственным и универсальным воплощением понятия компонента в языке C++ является пространство имен. Еще раз отметим, что не все элементы компонента являются классами.

В идеале компонент должен описываться набором интерфейсов, которые он использует для собственной реализации, и набором интерфейсов, которые он предоставляет своим пользователям. Все прочее — это внутренние детали реализации, которые должны скрываться от «внешнего мира». Набор интерфейсов — это термины, в которых компонент описывает его проектировщик. Программист же вынужден отображать интерфейсы в объявления. Классы и иерархии классов определяют интерфейсы, а пространства имен позволяют сгруппировать их в наборы, которые компонент использует сам и которые он предоставляет пользователям:



При использовании техники, описанной в §8.2.4.1, это превращается в следующее:

```
namespace A
{
  // ...
}

namespace X                    // интерфейс компонента X
{
  using namespace A;           // зависит от объявлений из A
  void f();
}

namespace X_impl               // средства, нужные реализации X
{
  using namespace X;
  // ...
}

void X::f()
{
  using namespace X_impl;      // зависит от объявлений из X_impl
  // ...
}
```

Основной интерфейс *X* компонента не должен зависеть от интерфейса реализации *X_impl*.

Компонент может содержать множество классов, не предназначенных для общего использования. Такие классы следует прятать внутри классов реализации или пространств имен:

```
namespace X_impl
{
  class Widget
  {
    // ...
  };
  // ...
}
```

Это гарантирует, что *Widget* не может использоваться из других частей программы. Однако классы, которые отражают общеполезные понятия, являются хорошими кандидатами для многократного повторного использования, и их желательно включать в интерфейс компонента:

```
class Car
{
  class Wheel
  {
    // ...
  };

  Wheel flw, frw, rlw, rrw;
  // ...
```

```
public:
  // ...
};
```

Во многих случаях колеса лучше спрятать ради чистоты абстрактной концепции автомобиля (нельзя использовать колеса в отрыве от главного понятия — автомобиля). Однако класс колес ***Wheel*** сам по себе кажется вполне пригодным для самостоятельного использования, так что возможно его все же лучше вынести из класса ***Car***:

```
class Wheel
{
  // ...
};

class Car
{
  Wheel flw, frw, rlw, rrw;
  // ...
public:
  // ...
};
```

Решение оставить класс внутри или вынести наружу (оба способа широко применяются на практике) зависит от конкретных целей проекта и совокупности понятий, составляющих проект. По умолчанию класс локализуют как только могут до тех пор, пока не появляются серьезные причины сделать его более доступным.

У интересных функций и данных есть отвратительная тенденция всплывать в глобальное пространство имен, в широко используемые пространства имен или корневой класс иерархии. Это может ненароком приоткрыть детали реализации и привести к проблемам, характерным для глобальных функций и данных. Вероятность этого увеличивается в случае однокоренных иерархий и для проектов с малым числом пространств имен. В контексте классовых иерархий с этим можно бороться с помощью виртуальных базовых классов (§15.2.4). Проблема же малого числа пространств имен решается введением множества небольших реализационных пространств имен.

Заметьте, что заголовочные файлы обеспечивают мощный механизм предоставления разных видов на один и тот же компонент для разных групп пользователей, а также для устранения пользовательской информации о классах реализации (§9.3.2).

24.4.1. Шаблоны

С проектной точки зрения шаблоны служат двум, слабо связанным между собой целям:

- Обобщенному программированию.
- Политике параметризации.

На ранних стадиях проектирования операции — это просто операции. Но на более поздних стадиях, когда появляется необходимость описать типы их операндов, проявляется особая важность шаблонов для таких статически типизированных язы-

ков программирования, как С++. Без шаблонов стали бы повторяться определения функций, а иначе проверка типов была бы отнесена на стадию выполнения программ (§24.2.3). Наилучшим кандидатом на роль шаблонов являются операции, реализующие универсальные алгоритмы, пригодные для разных типов. Если все такие операнды относятся к одной иерархии классов, и особенно если нужно добавлять типы операндов на стадии выполнения программы, тогда тип операнда лучше всего представить классом (часто абстрактным классом). Если же типы операндов не вписываются в единственную иерархию классов и если особо важна эффективность, операцию лучше всего реализовать в виде шаблона. Стандартные контейнеры и поддерживающие их алгоритмы иллюстрируют ситуацию, когда несвязанные между собой типы операндов и требование высокой эффективности вынуждают использовать шаблоны (§16.2).

Чтобы предметнее проиллюстрировать альтернативу шаблоны/иерархия рассмотрим, как можно обобщить простую итерацию:

```
void print all (Iter_for_T x)
{
  for (T* p = x.first() ; p; p = x.next() ) cout << *p;
}
```

Здесь предполагается, что ***Iter_for_T*** обеспечивает операции, возвращающие ***T****.

Мы можем сделать итератор ***Iter_for_T*** параметром шаблона:

```
template<class Iter_for_T> void print_all (Iter_for_T x)
{
  for (T* p = x.first() ; p; p = x.next() ) cout << *p;
}
```

Это позволяет нам использовать самые разные итераторы до тех пор, пока они предоставляют операции ***first***() и ***next***() надлежащего смысла, и если во время компиляции мы знаем типы итераторов для каждого вызова ***print_all***(). Стандартные контейнеры и алгоритмы базируются на этой идее.

С другой стороны, мы можем трактовать ***first***() и ***next***() как интерфейс к итераторам, и определить класс, представляющий этот интерфейс:

```
class Iter
{
public:
  virtual T* first() const = 0;
  virtual T* next() = 0;
};

void print_all2 (Iter& x)
{
  for (T* p = x.first() ; p; p = x.next() ) cout << *p;
}
```

Мы можем пользоваться любыми итераторами, производными от ***Iter***. Как видно, реальный клиентский код не зависит от того, используем ли мы для аргументов параметры шаблона или классовую иерархию — различаются лишь эффективность и нюансы, связанные с перекомпиляцией. В частности, вот конкретный пример использования класса ***Iter*** в качестве аргумента шаблона:

```
void f(Iter& i)
{
  print_all(i);        // используется шаблон
  print_all2(i);
}
```

Следовательно, два рассматриваемых подхода могут и дополнять друг друга.

Часто шаблону требуются функции и классы в качестве части своей реализации. Многие из них для сохранения универсальности и эффективности сами должны быть шаблонами. В этом смысле алгоритмы становятся обобщенными по множеству типов. Такой стиль применения шаблонов называется *обобщенным программированием* (*generic programming*) (§2.7). Когда мы вызываем **std::sort()** для контейнера **vector**, элементы вектора становятся операндами операции **sort()**; можно сказать, что алгоритм **sort()** является обобщенным по типу элементов вектора. Кроме того, этот алгоритм является обобщенным и по типам контейнеров, поскольку задействуются лишь итераторы произвольных стандартных контейнеров (§16.3.1).

Еще алгоритм **sort()** параметризован по критерию сравнения (§18.7.1). С точки зрения проектировщика это отличается от простой параметризации операции по типам ее аргументов. Решение параметризовать алгоритм таким образом, чтобы влиять на особенности работы алгоритма, является проектным решением более высокого уровня. Тем самым проектировщик/программист получают дополнительный канал влияния на определение политики выполнения некоторых действий в алгоритме. Однако с точки зрения языка программирования никаких особенностей тут нет.

24.4.2. Интерфейсы и реализации

Идеальный интерфейс:

- предоставляет пользователю полный и согласованный набор концепций,
- согласован для всех частей компонента,
- не открывает деталей реализации пользователям,
- может реализовываться разными способами,
- статически типизирован,
- выражается с помощью типов уровня приложения,
- зависит ограниченно и строго определенным образом от других интерфейсов.

Отметив необходимость согласованности классов, представляющих интерфейс компонента остальному миру (§24.4), мы можем теперь упростить изложение, ограничившись единственным классом:

```
class Y {/*...*/};          // нужен для X
class Z {/*...*/};          // нужен для X

class X                     // пример плохого стиля интерфейса
{
  Y a;
  Z b;

public:
  void f(const char* ...);
```

```
    void g (int[], int);
    void set_a (Y&);
    Y& get_a ();
};
```

У этого интерфейса есть несколько потенциальных проблем:

- Он использует типы *X* и *Y* таким образом, что для компиляции требуется знать их объявления.
- Функция *X*::*f*() принимает произвольное число аргументов неизвестного типа (возможно, под управлением «форматирующей строки», передаваемой в качестве первого аргумента; §21.8).
- Функция *X*::*g*() принимает аргумент типа *int*[]. Это допустимо, но означает низкий уровень абстракции. Массив целых чисел не описывает себя сам, и неясно, сколько элементов он содержит.
- Функции *set_a*() и *get_a*() с большой вероятностью открывают детали представления класса *X*, разрешая прямой доступ к *X*::*a*.

Эти функции-члены предоставляют слишком низкоуровневый интерфейс. Обычно, классы со столь низкоуровневыми интерфейсами уместны в качестве деталей реализации частей более крупных компонентов (если вообще где-либо уместны). В идеале, функции интерфейса должны принимать самоописываемые аргументы.

Язык C++ разрешает программисту экспонировать часть реализации класса в качестве части интерфейса. Эта часть реализации может быть закрытой (private или protected), но она доступна компилятору, чтобы он мог выделять память под автоматические переменные, встраивать функции и т.д. Отрицательным эффектом такого решения является возможность порождения нежелательных зависимостей в случае, когда пользовательские типы (классы) присутствуют в реализации класса. На самом деле, порождают ли члены класса с типами *Y* и *Z* проблемы или нет, зависит от того, какими на самом деле типами являются *Y* и *Z*. Если это простые типы, вроде ***list***, ***complex*** и ***string***, их применение чаще всего вполне оправдано. Это весьма стабильные типы, а необходимость включения их объявлений не является чрезмерной нагрузкой для компилятора. Однако если *Y* и *Z* сами являются интерфейсными классами больших и значимых компонентов, таких как графическая система или система управления банковскими счетами, неразумно зависеть от них столь непосредственно. В подобных случаях лучше использовать указатели или ссылки:

```
class Y;
class Z;

class X    // X имеет доступ к Y и Z только через указатели и ссылки
{
    Y* a;
    Z& b;
    // ...
};
```

Это разъединяет определение *X* от определений *Y* и *Z* (остается зависимость *X* от имен *Y* и *Z*). Реализация *X* будет, конечно же, по-прежнему зависеть от определений *Y* и *Z*, но это напрямую не относится к пользователям класса *X*.

Это иллюстрирует важное положение: интерфейс, скрывающий значительную часть информации — как и должен делать полезный интерфейс — имеет намного меньше зависимостей, чем скрываемая им реализация. Например, определение класса *X* можно откомпилировать без доступа к определениям *Y* и *Z*. Однако определения функций-членов класса *X*, манипулирующие объектами *Y* и *Z*, нуждаются в определениях *Y* и *Z*. При анализе зависимостей зависимости интерфейса и реализации должны рассматриваться отдельно. В обоих случаях для упрощения системы желательно иметь ациклические графы зависимостей. Этот идеал гораздо более важен и достижим для интерфейсов, чем для реализации.

Отметим, что класс может определить три интерфейса:

```
class X
{
private:
  // доступен только для членов и друзей
protected:
  // доступен членам и друзьям, а также
  // членам и друзьям производных классов
public:
  // доступен широкой публике
};
```

Кроме того, друзья тоже относятся к открытому интерфейсу (§11.5).

Поля данных должны попадать в интерфейс в самом редком случае. Они должны быть закрытыми, если нет веских причин делать их более доступными. В последнем случае их желательно делать защищенными, если нет конкретных причин, по которым они должны стать открытыми. Функции и типы, входящие в открытый интерфейс класса, должны соответствовать концепции, которую класс представляет.

Отметим еще, что для большей степени сокрытия представления можно использовать абстрактные классы (§2.5.4, §12.3, §25.3).

24.4.3. «Жирные» интерфейсы

В идеале интерфейс должен предоставлять лишь те операции, которые имеют смысл и хорошо представимы любым производным классом, реализующим этот интерфейс. Это, однако, не всегда легко сделать. Рассмотрим списки, массивы, ассоциативные массивы, деревья и т.д. Как показано в §16.2.2, соблазнительно и иногда полезно обобщить эти типы до типа *контейнер* (*container*), который можно было бы использовать в качестве интерфейса ко всем перечисленным структурам данных. Как кажется, это могло бы освободить пользователя от необходимости работать с деталями многочисленных специализированных контейнеров. Однако определить интерфейс универсального контейнера не так-то легко. Допустим, мы хотим определить ***Container*** как абстрактный тип. Какими операциями его нужно снабдить? Мы можем ввести лишь операции, которые допустимы для любого из перечисленных специализированных контейнеров, то есть фактически пересечение множеств их операций (до смешного узкий интерфейс). Более того, во многих практических случаях такое пересечение представляет собой пустое множество.

В качестве иной крайности мы можем ввести объединение множество всех операций и выдавать сообщение об ошибке, если во время выполнения программы через такой интерфейс произошло обращение к операции, не поддерживаемой объектом. Интерфейс, реализующий объединение интерфейсов ко множеству концепций, называют *жирным интерфейсом* (*fat interface*). Рассмотрим универсальный контейнер объектов типа *T*:

```
class Container
{
public:
  struct Bad_oper      // класс исключений
  {
    const char* p;
    Bad_oper(const char* pp) : p(pp) { }
  };

  virtual void put(const T*) {throw Bad_oper("Container::put");}
  virtual T* get() {throw Bad_oper("Container::get");}

  virtual T*& operator[](int) {throw Bad_oper("Container::[](int)");}
  virtual T*& operator[](const char*) {throw Bad_oper("Container::[](char*)");}
  // ...
};
```

Тогда конкретные контейнеры можно объявлять следующим образом:

```
class List_container: public Container, private list
{
public:
  void put(const T*);
  T* get();
  // ... отсутствует operator[] ...
};

class Vector_container: public Container, private vector
{
public:
  T*& operator[](int);
  T*& operator[](const char*);
  // ... отсутствуют put() и get() ...
};
```

Пока соблюдаются предосторожности, все в порядке:

```
void f()
{
  List_container sc;
  Vector_container vc;
  // ...
  user(sc, vc);
}

void user(Container& c1, Container& c2)
{
  T* p1 = c1.get();
```

```
    T* p2 = c2[3];
    // не используйте c2.get() или c1[3]
    // ...
}
```

Однако немногие структуры данных одновременно поддерживают и индексацию, и операции, характерные для списков. Поэтому идея объединить эти операции в одном интерфейсе вряд ли хороша. Это влечет либо интенсивный опрос типов во время выполнения (§15.4), либо приводит к генерации исключений (глава 14) ради устранения ошибок времени исполнения. Например:

```
void user2(Container& c1, Container& c2)    // обнаружение просто, восстановление - нет
{
  try
  {
    T* p1 = c1.get();
    T* p2 = c2[3];
    // ...
  }
  catch(Container::Bad_oper& bad)
  {
    // Oops!
    // И что теперь делать?
  }
}
```

или

```
void user3(Container& c1, Container& c2)  // раннее обнаружение не просто;
                                          // восстановление также не просто
{
  if(dynamic_cast<List_container*>(&c1) &&
     dynamic_cast<Vector_container*>(&c2))
  {
    T* p1 = c1.get();
    T* p2 = c2[3];
    // ...
  }
  else
  {
    // Oops!
    // И что теперь делать?
  }
}
```

В обоих случаях может пострадать быстродействие и код может быть излишне большим. В результате, программисты могут понадеяться на лучшее, то есть что пользователи будут работать с программой надлежащим образом и ошибок возникать не будет. К тому же исчерпывающее тестирование в данном случае труднодостижимо и слишком дорого.

Следовательно, жирных интерфейсов лучше избегать, когда производительность является важнейшим условием, когда требуется высокая надежность исполнения

и вообще, когде есть другая альтернатива. Применение жирных интерфейсов ослабляет степень соответствия между концепциями проекта и классами, и открывает простор для использования наследования исключительно ради удобства реализации.

24.5. Советы

1. Эволюционируйте в направлении абстракции данных и объектно-ориентированного программирования; §24.2.
2. Применяйте средства языка C++ и технологии программирования сообразно необходимости; §24.2.
3. Стремитесь к соответствию стиля программирования стилю проектирования; §24.2.1.
4. В проектировании фокусируйтесь на концепциях/классах, а не на функциях/вычислениях; §24.2.1.
5. Применяйте классы для представления концепций; §24.2.1, §24.3.
6. Применяйте наследование исключительно для отражения иерархических отношений между концепциями; §24.2.2, §24.2.5, §24.3.2.
7. Базируйте строгие гарантии касательно интерфейсов в терминах статических типов уровня приложения; §24.2.3.
8. Для решения четко определенных задач применяйте генераторы кода и другие средства визуального программирования; §24.2.4.
9. Избегайте тех генераторов кода и средств визуального программирования, что плохо стыкуются с языками программирования общего назначения; §24.2.4.
10. Не смешивайте между собой разные уровни абстракции; §24.3.1.
11. Фокусируйтесь на проектировании компонентов; §24.4.
12. Убедитесь, что виртуальные функции имеют четко определенный смысл, и что все замещающие функции реализуют надлежащее поведение; §24.3.4, §24.3.2.1.
13. Используйте открытое наследование для выражения отношения "is-a"; §24.3.4.
14. Используйте агрегацию для отражения отношения "has-a"; §24.3.4.
15. Для отражения отношения включения используйте поля данных с типом включаемого объекта, а не с типом указателя на этот объект; §24.3.3, §24.3.4.
16. Старайтесь делать отношения использования ясными и понятными, нециклическими и минимальными; §24.3.5.
17. Определяйте инварианты для всех классов; §24.3.7.1.
18. Предусловия, постусловия и иные утверждения выражайте с помощью утверждений (возможно, с помощью *Assert*()); §24.3.7.2.
19. Определяйте интерфейсы, раскрывая минимум необходимой информации; §24.4.

20. Минимизируйте зависимости интерфейса от других интерфейсов; §24.4.2.
21. Поддерживайте строгую типизацию интерфейсов; §24.4.2.
22. Выражайте интерфейсы на уровне типов приложения; §24.4.2.
23. Выражайте интерфейс таким образом, чтобы запрос мог передаваться на удаленный сервер; §24.4.2.
24. Избегайте «жирных» интерфейсов; §24.4.3.
25. Применяйте закрытые поля данных и функции-члены где только можно; §24.4.2.
26. Применяйте альтернативу ***private/protected*** для различения нужд разработчиков производных классов и обычных пользователей; §24.4.2.
27. Для обобщенного программирования применяйте шаблоны; §24.4.1.
28. Пользуйтесь шаблонами для параметризации политики выполнения алгоритмов; §24.4.1.
29. Применяйте шаблоны, если нужно, чтобы разрешение типов производилось на этапе компиляции; §24.4.1.
30. Используйте иерархии классов, когда разрешение типов необходимо на этапе выполнения; §24.4.1.

25

Роли классов

Кое-что можно и поменять,
но фундаментальные вопросы должны быть незыблемыми.
— Стивен Дж. Гульд

Разновидности классов — конкретные типы — абстрактные типы — узловые классы — изменение интерфейсов — объектный ввод/вывод — операции — интерфейсные классы — дескрипторные классы — прикладные среды разработки — советы — упражнения.

25.1. Разновидности классов

Класс в языке C++ — это конструкция, которая отвечает разнообразным проектным целям. Я нахожу, что многие запутанные проблемы проектирования часто решаются введением нового класса, представляющего понятие, которое в предыдущем варианте проекта не было выявлено явным образом (при этом иногда приходится и удалять классы). Множество ролей классов в проекте приводит ко множеству видов классов в программе, специально приспособленных для решения конкретных задач. В настоящей главе рассматривается несколько классов-архетипов с присущими им достоинствами и недостатками:

- §25.2 Конкретные типы
- §25.3 Абстрактные типы
- §25.4 Узлы
- §25.5 Операции
- §25.6 Интерфейсы
- §25.7 Дескрипторные классы (handles)
- §25.8 Прикладные среды разработки

Эти разновидности классов являются проектными понятиями, а не языковыми конструкциями. В идеале (наверное, недостижимом), хорошо бы иметь минималь-

ный набор простых и ортогональных друг другу классов, из которых можно было составлять любые полезные и эффективно работающие классы. Отметим, что в проектах могут пригодиться любые из этих разновидностей классов, и ни одна из них по своей природе не лучше других в общем случае. Большая путаница при обсуждении вопросов проектирования и программирования исходит от людей, предпочитающих лишь одну или две разновидности классов. Делается это, якобы, ради простоты, но на деле приводит лишь к неуклюжему и неестественному применению любимых разновидностей классов.

Мы здесь будем рассматривать чистые формы перечисленных разновидностей классов, хотя можно использовать и гибридные формы. Гибрид, однако, должен появляться в результате осознанного проектного решения, а не в результате отказа от четкого выбора. Откладывание решения выбора на потом часто является завуалированной формой «отказа думать». Новичкам лучше держаться подальше от гибридов и придерживаться стилей уже существующих компонентов, имеющих характеристики, похожие на те, что требуется получить от нового решения. Только опытным программистам по плечу писать компоненты и библиотеки общего назначения, предназначенные для длительного использования, и которые нужно хорошо документировать и поддерживать все это время. См. §23.5.1.

25.2. Конкретные классы (типы)

Такие классы, как ***vector*** (§16.3), ***list*** (§17.2.2), ***Date*** (§10.3) и ***complex*** (§11.3, §22.5), являются *конкретными* (*concrete*) в том смысле, что каждый из них представляет относительно простое понятие вместе с операциями, достаточными для поддержки этого понятия. Кроме того, в каждом из этих классов имеется взаимнооднозначное соответствие между интерфейсом и реализацией, и ни один из этих классов не предназначен для создания производных классов. Как правило, конкретные классы не вписываются в иерархии связанных классов. Каждый конкретный тип понятен сам по себе с минимальнейшей отсылкой к другим классам. Если конкретный тип хорошо реализован, то использующие его программы столь же малы по размеру и столь же эффективны, как и варианты программы, в которых программист самостоятельно вручную кодирует специализированные версии той же самой концепции. Аналогично, если реализация изменяется существенно, изменяется и интерфейс для отражения этих изменений. Во всем этом конкретный класс напоминает встроенные типы (естественно, все встроенные типы конкретны). Определяемые пользователем конкретные типы, такие как комплексные числа, матрицы, сообщения об ошибках и т.д., часто представляют собой фундаментальные понятия для определенных предметных областей.

Точная природа классового интерфейса определяет, какие изменения в реализации имеют существенное значение в данном контексте; более абстрактные интерфейсы оставляют больше простора для изменения в реализации, но могут приводить к снижению быстродействия. Хорошая реализация не зависит от других классов больше, чем это необходимо, в результате чего снижаются затраты на «притирку» с иными классами как во время компиляции программы, так и во время ее выполнения.

Таким образом, подводя итог, можно сказать, что конкретный класс нацелен на то, чтобы:

1. Как можно точнее соответствовать частной концепции и стратегии ее реализации.
2. Обеспечивать затраты памяти и быстродействие, сравнимые с таковыми со специализированными «ручными» вариантами кода, за счет применения встраивания и операций, использующих все конкретные особенности концепции и ее реализации.
3. В минимальной степени зависеть от других классов.
4. Быть понятным и пригодным к использованию независимо от других классов.

В результате достигается тесная связь между кодом реализации и кодом пользователя класса. Если реализация как-то меняется, пользовательский код придется перекомпилировать, поскольку пользовательский код почти всегда содержит обращение ко встраиваемым функциям или внутренним переменным конкретного класса.

Название «конкретный тип» было выбрано по контрасту с распространенным термином «абстрактный тип». Взаимосвязь между конкретными и абстрактными типами обсуждается в §25.3.

Конкретные типы непосредственным образом не выражают отношения общности. Например, ***list*** и ***vector*** предоставляют схожие наборы операций и могут взаимозаменяемо использоваться в шаблонах функций. Тем не менее, никакого отношения общности у классов ***list<int>*** и ***vector<int>***, или между ***list<Shape*>*** и ***list<Circle*>*** (§13.6.3), нет, хотя мы сами-то и можем это сходство углядеть.

Для наивно спроектированных конкретных типов это приводит к тому, что код, использующий их схожим образом, будет выглядеть по-разному. Например, итерации по списку ***List*** при помощи функции ***next()*** совсем не похожи на итерации по типу ***Vector*** при помощи индексации:

```
void my (List& sl)
{
  for (T* p = sl.first() ; p; p = sl.next() )     // естественная итерация по списку
  {
    // мой код
  }
  // ...
}

void your (Vector& v)
{
  for (int i = 0; i<v.size() ; i++)               // естественная итерация по вектору
  {
    // ваш код
  }
  //...
}
```

Разница в стиле итераций вполне естественна в том смысле, что операция «получить следующий элемент» органична для списков, но не для векторов, а опера-

ция индексации является фундаментальной для векторов (но не для списков). Наличие органичных для выбранной стратегии реализации контейнера операций критически важно с точки зрения достижения высокой эффективности и удобства записи.

Неприятность такого подхода состоит в том, что программы для фундаментально схожих операций, используемых в двух приведенных выше циклах, выглядят различно, и код, применяющий конкретные типы для логически эквивалентных операций, не взаимозаменяем. В итоге, в программах реальных размеров выявить такие схожести бывает очень сложно, и сложно бывает перепроектировать систему для эксплуатации обнаруженной общности. Удачным примером перепроектирования, позволившим задействовать сходство между конкретными типами, не пожертвовав при этом ни элегантностью, ни эффективностью, являются стандартные контейнеры и алгоритмы (§16.2).

Если функция принимает аргумент конкретного типа, уже не будет никакой возможности принять менее специфичный аргумент (как это бывает в отношениях наследования). Поэтому попытка задействовать сходство между конкретными типами с неизбежностью приводит к использованию шаблонов и обобщенного стиля программирования, как описано в §3.8. Для стандартной библиотеки итерации принимают следующий вид:

```
template<class C> void ours (C& c)
{
  for (typename C::iterator p=c.begin(); p!=c.end(); ++p) // итерации стандарт. биб-ки
  {
    // ...
  }
}
```

Здесь использовано фундаментальное сходство между контейнерами, открывающее дополнительные возможности, используемые стандартными алгоритмами (глава 18).

Чтобы эффективно применять конкретные типы, пользователь должен хорошо понимать его частные детали. В библиотеке конкретных классов нет никакого общего свойства для всех содержащихся в ней типов, так что пользователю приходится вникать в каждый класс по отдельности. Это плата за быстродействие и компактность, причем эта плата может окупиться, а может и не окупиться. В то же время, часто легче понять и использовать индивидуальный конкретный класс, чем более общий и абстрактный. Это характерно для таких широко известных типов данных, как, например, массивы и списки.

В идеале лучше скрывать реализацию в максимально возможной степени, не допуская при этом серьезных ухудшений производительности. Здесь встраиваемые функции могут дать большой выигрыш.

Открывать поля данных класса, чтобы пользователь манипулировал ими напрямую или через специально предназначенные для этого функции-члены, редко когда бывает хорошей идеей (§24.4.2). Конкретные типы должны оставаться типами, а не мешками битов с несколькими функциями, добавленными лишь для удобства.

25.2.1. Многократное использование конкретных типов

Конкретные типы редко когда бывает полезным использовать в виде базовых классов. Каждый конкретный тип призван обеспечить четкое и эффективное представление единственной концепции. Маловероятно, чтобы класс, хорошо справляющийся с такой задачей, был хорошим кандидатом для создания других, родственных классов посредством открытого наследования. Такие классы лучше использовать в качестве типов полей данных других классов, или в качестве закрытых базовых классов. Тут они могут использоваться эффективно, не смешивая при этом свои интерфейсы и реализацию с интерфейсами и реализацией новых классов. Рассмотрим наследование от класса ***Date***:

```
class My_date : public Date
{
  // ...
};
```

Нужно ли когда-нибудь использовать ***My_date*** вместо ***Date***? Это, конечно, зависит от того, что именно представляет собой ***My_date***, но по моему опыту конкретные классы редко бывают полезны в качестве базовых без серьезных модификаций.

Немодифицируемым образом конкретные классы «повторно используются» так же, как и встроенные типы, вроде ***int*** (§10.3.4). Например:

```
class Date_and_time
{
private:
  Date d;
  Time t;

public:
  // ...
};
```

Такая форма использования (повторного?) проста, эффективна и продуктивна.

Может, это ошибка проектирования привела к тому, что класс ***Date*** не пригоден к повторному использования посредством наследования от него производных классов? Иногда слышны утверждения, что *каждый класс* должен проектироваться с учетом последующей модификации посредством замещения и посредством использования его членов функциями-членами производных классов. Эта точка зрения привела бы к следующему варианту класса ***Date***:

```
class Date2
{
public:
  // публичный интерфейс, состоящий преимущественно из виртуальных функций

protected:
  // другие детали реализации (возможно, включая часть представления)

private:
  // представление и иные детали реализации
};
```

Часть представления класса сделана защищенной с целью облегчить замещение функций в производных классах. Это позволяет сделать тип ***Date2*** весьма податливым к модификациям посредством наследования, оставляя его открытый пользовательский интерфейс неизменным. Цена такого решения следующая:

1. *Базовые операции менее эффективны.* Обращение к виртуальным функциям в C++ немного медленнее, чем к обычным функциям; виртуальные функции редко когда бывает разумным делать встраиваемыми; объекты классов с виртуальными функциями требуют добавочного расхода памяти.
2. *Необходимость динамического использования свободной памяти.* Цель введения класса ***Date2*** — позволить использовать взаимозаменяемым образом объекты классов, производных от ***Date2***. Поскольку размеры этих объектов разные, их нужно очевидным образом размещать в динамически выделяемой свободной памяти, обращаться к которой придется с помощью указателей или ссылок. Локализация данных класса при этом резко уменьшается.
3. *Неудобство для пользователей.* Выгоды от полиморфизма, обеспечиваемого виртуальными функциями, достигаются лишь при манипулировании объектами класса через указатели или ссылки.
4. *Слабая инкапсуляция.* Виртуальные функции можно замещать, и защищенными данными можно манипулировать из производных классов (§12.4.1.1).

Рассмотренная цена решения в общем случае не столь уж и велика, и часто это решение оказывается именно тем, что нужно (§25.3, §25.4). В то же время, для таких простых типов, как ***Date2***, цена явно излишняя и совершенно необязательная.

Наконец, четко спроектированный конкретный тип часто подходит для реализации (представления) более гибких (податливых) типов. Например:

```
class Date3
{
public:
  // публичный интерфейс, состоящий преимущественно из виртуальных функций

private:
  Date d;
};
```

Именно этот способ следует использовать при необходимости встроить конкретные типы в иерархию классов. См. также §25.10[1].

25.3. Абстрактные типы

Простейшим способом ослабить связь между пользователями класса и его разработчиками, а также между создающим объекты кодом и кодом, использующим объекты, является введение абстрактного класса, предоставляющего интерефейс к набору реализаций некоторой общей концепции. Рассмотрим шаблон ***Set*** (множество):

```
template<class T> class Set
{
public:
```

```
    virtual void insert(T*) = 0;
    virtual void remove(T*) = 0;

    virtual int is_member(T*) = 0;

    virtual T* first() =0;
    virtual T* next() = 0;

    virtual ~Set() {}
};
```

Это интерфейс ко множеству со встроенной концепцией итерации по его элементам. Типично отсутствие конструкторов и присутствие виртуального деструктора (§12.4.2). Допускаются разные реализации (§16.2.1). Например:

```
template<class T> class List_set: public Set<T>, private list<T>
{
  // ...
};

template<class T> class Vector_set: public Set<T>, private vector<T>
{
  // ...
};
```

Абстрактный класс предоставляет интерфейс к любым реализациям. Это означает, что мы можем применять ***Set***, не зная, какая из реализаций используется. Например:

```
void f{Set<Plane*>& s)
{
  for (Plane** p = s.first(); p; p = s.next())
  {
    // мой код
  }
  // ...
}

List_set<Plane*> sl;
Vector_set<Plane*> v(100);

void g()
{
  f(sl);
  f(v);
}
```

Для конкретных типов требуется перепроектирование классов реализации с целью отражения общности и применение шаблонов для эксплуатации этой общности. Здесь же общность отражается интерфейсом (то есть ***Set***), а от классов реализации не требуется никакой другой общности, кроме способности реализовывать интерфейс.

Более того, пользователи ***Set*** не знают объявлений ***List_set*** и ***Vector_set***, и поэтому пользователи не зависят от этих объявлений и не нуждаются в перекомпиляции в случае, если изменяются ***List_set*** или ***Vector_set***, или даже если вводится новая реа-

лизация ***Set*** — скажем, ***Tree_set***. Все зависимости сосредоточены в функциях, которые явным образом используют классы, производные от ***Set***. В частности, предполагая традиционное использование заголовочных файлов, программисту, пишущему ***f(Set&)***, нужно включить лишь ***Set.h***, а не ***List_set.h*** или ***Vector_set.h***. «Реализационный» заголовочный файл нужен лишь там, где создаются ***List_set*** или ***Vector_set***. Пользовательский код можно еще сильнее изолировать от классов реализации, введя абстрактный класс, обрабатывающий запросы на создание объектов (так называемая «фабрика»; §12.4.4).

Отделение интерфейса от реализации означает отсутствие доступа к операциям, свойственным конкретной реализации, но не достаточно общим, чтобы стать частью интерфейса. Например, поскольку множество неупорядочено по своей сути, мы не можем включить в интерфейс операцию индексации даже в том случае, когда для реализации применяется частное представление в виде массива. Отсюда вытекает некоторая потеря эффективности за счет запрета ручной оптимизации. Кроме того, встраивание функций, как правило, недостижимо (кроме локального контекста, когда компилятор знает истинный тип), а все интересные операции интерфейса представлены виртуальными функциями. Как и в случае с конкретными типами, затраты на абстрактные типы иногда окупаются, а иногда нет. Подводя итог, можно сказать, что абстрактные типы предназначены для того, чтобы:

1. Определять одну концепцию с разными реализациями, сосуществующими в пределах программы.
2. Обеспечивать приемлемые быстродействие и затраты памяти посредством использования виртуальных функций.
3. Минимизировать зависимость каждой реализации от других классов.
4. Иметь возможность самоописания.

Абстрактные типы не лучше конкретных типов; они просто другие. Разработчики библиотек часто предоставляют и те, и другие типы, оставив выбор за пользователем. Попытки ограничить общность абстрактного типа ради конкуренции с конкретными типами в быстродействии как правило оканчиваются неудачей; это компрометирует возможность использовать взаимозаменяемые реализации без значительной перекомпиляции после их модификации. Также неудачей оканчиваются и попытки увеличить универсальность конкретных типов; при этом снижается их эффективность и удобство использования. Конкретные и абстрактные типы могут сосуществовать — на самом деле они *должны* сосуществовать, ибо конкретные типы обеспечивают реализацию абстрактных типов. Однако не следует смешивать эти типы хаотическим образом.

Часто, абстрактные типы не предназначены для порождения производных классов, отличных от классов непосредственной реализации. Однако из абстрактного класса можно сконструировать новый интерфейс, породив расширенный производный класс. Этот новый интерфейс (новый абстрактный класс) также нужно реализовать посредством неабстрактных производных классов (§15.2.5).

Почему же мы сразу не породили ***List*** и ***Vector*** от ***Set***, избежав введения классов ***List_set*** и ***Vector_set***? Иными словами, зачем нам потребовались конкретные типы вместо абстрактных типов? Здесь нужно учитывать следующие соображения:

1. *Эффективность.* Нам могут потребоваться конкретные типы, вроде ***vector*** и ***list***, без затрат на отделение интерфейса от реализации (сопутствуют стилю абстрактных классов).
2. *Повторное использование.* Нам нужен механизм приспособления спроектированных вовне типов (вроде ***vector*** и ***list***) в новые библиотеки или программы путем предоставления для них нового интерфейса (а не путем переписывания их кода).
3. *Множественные интерфейсы.* Использование одного класса в качестве базового для многих классов приводит к жирным интерфейсам (§24.4.3). Для новых задач лучше снабдить класс новым интерфейсом (вроде интерфейса ***Set*** для ***vector***), чем приспосабливать старый интерфейс под новые задачи.

Естественно, эти положения связаны между собой. Подробно они обсуждались на примере ***Ival_box*** (§12.4.2, §15.2.5), а также в контексте проектирования контейнеров (§16.2).

25.4. Узловые классы

Часто классовые иерархии используют наследование не в том ключе, как в случае абстрактного класса, предоставляющего интерфейс к реализациям. Здесь класс рассматривается в качестве фундамента, на котором строится часть системы. Даже, если этот класс абстрактный, он все равно имеет некоторое представление, обеспечивающее услуги своим производным классам. Примерами узловых классов служат ***Polygon*** (§12.3), первоначальный вариант ***Ival_slider*** (§12.4.1) и ***Satellite*** (§15.2).

В типичном случае, класс иерархии представляет собой некоторую концепцию, для которой производные классы могут рассматриваться как специализации этой концепции. *Узловой класс* (*node class*) *иерархии* использует сервисы базового класса для того, чтобы предоставлять свои собственные сервисы; для этого он вызывает функции-члены базового класса. Типичный узловой класс не только реализует интерфейс, описанный базовым классом (как это для абстрактного типа делает класс реализации), но и добавляет свои собственные функции, обеспечивая тем самым расширенный интерфейс. Рассмотрим класс ***Car*** из примера моделирования уличного движения в §24.3.2:

```
class Car : public Vehicle
{
public:
  Car (int passengers, Size_category size, int weight, int fc)
     : Vehicle (passengers, size, weight) , fuel_capacity (fc)  { /* . . . */ }

  // замещаем виртуальные функции из Vehicle:
  void turn (Direction) ;
  // ...
  // добавляем специфические автомобильные функции:
  virtual void add_fuel (int amount) ;        // заправка топливом
  //...
};
```

Здесь важны конструкторы, через которые программист задает основные моделируемые свойства, и виртуальные функции, позволяющие моделирующим уличное движение операциям манипулировать транспортным средством, не зная его точного типа. Класс ***Car*** можно использовать следующим образом:

```
void user ()
{
  // ...
  Car* p = new Car (3, economy, 1500, 60) ;
  drive (p, bs_home, MH) ;       // входим в общую схему имитации уличного движения
  // ...
}
```

Обычно узловому классу нужны нетривиальные конструкторы. Этим он отличается от абстрактных типов, которые редко когда имеют конструкторы.

Операции над классом ***Car*** в своих реализациях будут, как правило, пользоваться операциями базового класса ***Vehicle***. Кроме того, пользователь класса ***Car*** также рассчитывает на услуги базового класса. Например, базовый класс ***Vehicle*** предоставляет функции, имеющие дело с весом и размерами, так что классу ***Car*** нет нужды этим заниматься:

```
bool Bridge::can_cross (const Vehicle& r)
{
  if (max_weigh < r.weight () ) return false;
  // ...
}
```

Это позволяет программистам создавать из узлового класса ***Vehicle*** новые классы, такие как ***Car*** и ***Truck***, специфицируя и реализуя лишь то, что должно быть отличным от ***Vehicle***. Это часто называют «программированием отличий».

Как и многие узловые классы, класс ***Car*** является хорошей кандидатурой для порождения производных классов. Например, ***Ambulance*** (скорая помощь) нуждается в дополнительных данных и функциях, связанных с экстренностью службы:

```
class Ambulance : public Car, public Emergency
{
public:
  Ambulance () ;

  // замещаем виртуальные функции из Car:
  void turn (Direction) ;
  // ...

  // замещаем виртуальные функции из Emergency:
  virtual void dispatch_to (const Location&) ;
  // ...

  // добавляем специфические для Ambulance функции:
  virtual int patient_capacity () ;     // количество носилок
  // ...
};
```

Подводя итог, можно сказать, что узловой класс:

1. Опирается на базовые классы как с целью реализации, так и для предоставления услуг своим пользователям.
2. Обеспечивает более широкий интерфейс (с большим числом открытых функций-членов) по сравнению с базовыми классами.
3. В своем открытом интерфейсе опирается в первую очередь (но не исключительно) на виртуальные функции.
4. Зависит (прямо или косвенно) от всех своих базовых классов.
5. Может быть понят только в контексте своих базовых классов.
6. Полезен для дальнейшего порождения классов наследованием.
7. Используется для создания объектов.

Не все узловые классы будут удовлетворять пунктам 1, 2, 6 и 7, но большинство будет. Класс, не удовлетворяющий пункту 6, напоминает конкретный тип, так что его можно назвать *конкретным узловым классом* (*concrete node class*). Например, конкретный узловой класс может использоваться для реализации абстрактного класса (§12.4.2), а его объекты могут создаваться статически и в стеке. Такой класс также называют *листовым классом* (*leaf class*). Отметим, что любой код, оперирующий указателем или ссылкой на класс с виртуальными функциями, должен учитывать возможность появления дальнейших производных классов (или предположить, без прямой языковой поддержки, что дальнейших классов не будет). Класс, не удовлетворяющий пункту 7, напоминает абстрактный тип и может быть назван *абстрактным узловым классом* (*abstract node class*). По сложившейся традиции (неудачной) многие узловые классы имеют ряд защищенных членов, чтобы не слишком сильно ограничивать интерфейс для производных классов (§12.4.1.1).

Пункт 4 подразумевает, что для компиляции узлового класса программист должен включить объявления всех его прямых и непрямых базовых классов, а также все объявления, от которых зависят последние. В этом отношении узловой класс опять-таки отличается от абстрактных классов — пользователь последних не зависит от классов реализации и может не включать для компиляции своего кода информацию об этих классах.

25.4.1. Изменение интерфейсов

По определению, узловой класс является частью классовой иерархии. Не все классы иерархии должны предоставлять один и тот же интерфейс. В частности, производные классы могут предоставлять дополнительные функции-члены, а классы одного и того же уровня иерархии (sibling classes) — совершенно иной набор функций. С проектной точки зрения операция ***dynamic_cast*** (§15.4) представляется механизмом опроса объектов на предмет, обеспечивают ли они данный интерфейс.

Для примера рассмотрим простую систему ввода/вывода объектов. Пользователи хотят читать объекты из потока, определять, относятся ли они к ожидаемому типу, после чего использовать эти объекты. Например:

```
void user ()
{
  // открываем файл предположительно с фигурами типа Shape и прикрепляем
  // ss в качестве istream для этого файла
```

```
  //...
  Io_obj* p = get_obj(ss);      // читаем объект из потока
  if(Shape* sp = dynamic_cast<Shape*>(p))
  {
    sp->draw();                 // используем Shape
    //...
  }
  else
  {
    // oops: это не Shape
  }
}
```

Функция *user*() работает с фигурами исключительно посредством абстрактного класса *Shape* и может поэтому пользоваться любыми фигурами. Динамическое приведение типа посредством операции *dynamic_cast* здесь весьма существенно, ибо система объектного ввода/вывода может иметь дело со многими другими видами объектов, а пользователь может непреднамеренно открыть файл с объектами классов, о которых он никогда не слышал.

Система объектного ввода/вывода предполагает, что каждый записанный или считанный объект относится к классу, производному от *Io_obj*. Класс *Io_obj* должен быть полиморфным типом, чтобы разрешить нам применять операцию *dynamic_cast*. Например:

```
class Io_obj
{
public:
  virtual Io_obj* clone() const =0;
  virtual ~Io_obj() {}
};
```

Критически важной функцией в системе объектного ввода/вывода является функция *get_obj*(), которая считывает данные из потока *istream* и создает объекты на основании этих данных. Предположим, что данные, представляющие объект в потоке ввода, предваряются строкой, идентифицирующей класс объекта. Функция *get_obj*() должна прочесть этот строковый префикс и вызвать функцию, способную создать и инициализировать объект надлежащего класса. Например:

```
typedef Io_obj* (*PF)(istream&);
map<string,PF> iomap;

boot get_word(istream& is, string& s);   // читает слово из is в s

Io_obj* get_obj(istream& s)
{
  string str;
  bool b = get_word(s,str);               // читаем слово в str
  if(b==false) throw No_class();          // проблемы с форматом ввода/вывода

  PF f = io_map[str];                     // выбираем функцию по str
  if(f==0) throw Unknown_class();         // для str соответствия не найдено

  return f(s);                            // конструируем объект из потока
}
```

Ассоциативный массив типа ***map***, названный ***io_map***, содержит пары из имен классов и функций, способных создавать объекты этих классов.

Мы могли бы определить класс ***Shape*** обычным образом, кроме того, что его нужно сделать производным от ***Io_obj***, как того требует функция ***user()***:

```
class Shape : public Io_obj
{
  // ...
};
```

Однако интереснее (и реалистичнее) воспользоваться ранее определенным классом ***Shape*** (§2.6.2) без изменений:

```
class Io_circle : public Circle, public Io_obj
{
public:
  Io_circle* clone () const {return new Io_circle (*this) ; }
  Io_circle (istream&) ;          // инициализация из потока ввода
  static Io_obj* new_circle (istream& s) {return new Io_circle (s) ; }
  // ...
};
```

Этот пример показывает, насколько легче вписать класс в иерархию при помощи абстрактного класса, чем если бы мы в первую очередь рассматривали его в качестве узлового (§12.4.2, §25.3).

Конструктор ***Io_circle(istream&)*** инициализирует объект данными из своего аргумента типа ***istream***. Функция ***new_circle()*** помещается в ассоциативный массив ***io_map***, чтобы дать знать системе объектного ввода/вывода о новом классе. Например:

```
io_map ["Io_circle"] = &Io_circle: : new_circle;
```

Другие фигуры создаются таким же образом:

```
class Io_triangle : public Triangle, public Io_obj
{
  // ...
};
```

Это может и утомить, так что лучше определить шаблон:

```
template<class T> class Io : public T, public Io_obj
{
public:
  Io* clone () const {return new Io (*this) ; }      // замещаем Io_obj::clone()
  Io (istream&) ;                                    // инициализация из потока ввода
  static Io* new_io (istream& s) {return new Io (s) ; }
  //...
};
```

и уже с его помощью определить ***Io_circle***:

```
typedef Io<Circle> Io_circle;
```

Впрочем, нам по-прежнему нужно явно определить ***Io<Circle>::Io(istream&)***, поскольку при этом требуется знать подробности о ***Circle***.

Шаблон *Io* служит примером того, как вписать конкретные типы в классовую иерархию с помощью класса, являющегося узловым в этой иерархии. Шаблон наследует от типа его параметра, чтобы обеспечить приведение типа от *Io_obj*. К сожалению, из-за этого становится невозможным использовать *Io* со встроенными типами:

```
typedef Io<Date> Io_date;    // обертка для конкретного типа
typedef Io<int> Io_int;      // error: наследование от встроенного типа невозможно
```

Эту проблему можно уладить, введя отдельный шаблон для встроенных типов, или воспользовавшись классом, представляющим встроенный тип (§25.10[1]).

Эта простая система объектного ввода/вывода не может делать все на свете, но ее код почти умещается на одной странице, а ее ключевые механизмы имеют множество применений. Например, их можно использовать для вызова функции, строка с именем которой предоставляется пользователем, и для манипулирования объектами неизвестного типа при помощи интерфейсов, выявляемых средствами динамического определения типа.

25.5. Операции

Самый простой и наиболее очевидный способ определить какую-нибудь операцию в языке C++ — это написать соответствующую функцию. Однако если операция должна быть отложенной (delayed), или ее нужно передать куда-либо для выполнения, или она должна вовлекать в вычисления собственные данные или комбинироваться с другими операциями (§25.10[18,19]) и т.д. — то возникает желание оформить ее в виде класса, который может выполнить эту операцию, а также предоставить множество других услуг. Очевидным примером здесь являются классы функциональных объектов (классы объектов-функций), используемые совместно со стандартными алгоритмами (§18.4), а также манипуляторы, применяемые с потоками *iostream* (§21.4.6). В первом случае собственно действие вызывается операцией «круглые скобки», а во втором — операциями << и >>. В случае класса *Form* (§21.4.6.3) и *Matrix* (§22.4.7) классы-композиторы (compositor classes) позволяют отложить операцию до тех пор, пока не будет собрано достаточно информации для ее эффективного выполнения.

Простейшая форма класса-операции обычно содержит одну виртуальную функцию (называемую по традиции что-нибудь вроде «do_it» — «сделай это»):

```
struct Action
{
  virtual int do_it (int) =0;
  virtual ~Action () { }
};
```

Теперь мы можем написать код, например меню, который сможет сохранять действия (операции) для их последующего выполнения, не используя указателей на функции, не имея информации о вызванных объектах, и даже не зная имени операции, подлежащей выполнению. Например:

```
class Write_file : public Action
{
  File& f;
```

```
public:
  int do_it(int) {return f.write().succeed();}
};

class Error_response: public Action
{
  string message;

public:
  Error_response(const string& s) : message(s) { }
  int do_it(int);
};

int Error_response::do_it(int)
{
  Response_box db(message.c_str(), "continue", "cancel", "retry");

  switch(db.get_response())
  {
    case 0:
      return 0;
    case 1:
      abort();
    case 2:
      current_operation.redo();
      return 1;
  }
}

Action* actions[] = { new Write_file(f),
                      new Error_response("you blew it again"),
                      // ...
};
```

Пользователь ***Action*** может ничего не знать о производных классах, таких как ***Write_file*** и ***Error_response***.

Описанная техника чрезвычайно мощная и людям с привычкой к функциональной декомпозиции ею нужно пользоваться с известной осторожностью. Если слишком много классов начинают напоминать ***Action***, это может свидетельствовать об ухудшении проекта и крене в нечто излишне функциональное.

Наконец, класс может кодировать операцию для выполнения на другом сетевом (удаленном — remote) компьютере, или для сохранения ее на диске с целью выполнения когда-либо в будущем (§25.10[18]).

25.6. Интерфейсные классы

Одними из самых важных являются скромные интерфейсные классы, на которые часто смотрят свысока. Интерфейсный класс мало что делает- если бы делал, то не был бы тогда интерфейсным классом. Он просто приспосабливает подачу услуг под «местные» требования.

Чистейшая форма интерфейса даже не вызывает генерации кода. Рассмотрим специализацию шаблона ***Vector*** из §13.5:

```
template<class T> class Vector<T*> : private Vector<void*>
{
public:
  typedef Vector<void*> Base;

  Vector() {}
  Vector(int i) : Base(i) {}

  T*& operator[] (int i) {return reinterpret_cast<T*&>(Base::operator[] (i)); }
  // ...
};
```

Эта (частичная) специализация превращает небезопасный ***Vector<void*>*** в гораздо более полезное семейство векторных классов, безопасных с точки зрения типов. Встраиваемые функции позволяют сделать интерфейсные классы более приемлемыми. Например, здесь встраиваемая функция выполняет всего лишь подгонку типа, так что важно, что она не влечет дополнительных накладных расходов (ни по памяти, ни по производительности).

Естественно, абстрактный базовый класс, отражающий абстрактную концепцию и реализуемый конкретными типами (§25.2), тоже является в некотором смысле интерфейсным классом, как и дескрипторные классы из §25.7. Однако здесь мы фокусируемся на классах, которые не выполняют никаких специфических функций, кроме подгонки интерфейсов.

Рассмотрим проблему слияния двух иерархий с применением множественного наследования. Что делать, если имеет место конфликт имен, например, если виртуальные функции, выполняющие совершенно разные операции, имеют совпадающие имена? Рассмотрим, например, игру в «Дикий Запад», где взаимодействие с пользователем осуществляется посредством общего оконного класса:

```
class Window
{
  // ...
  virtual void draw();
};

class Cowboy
{
  // ...
  virtual void draw();
};

class Cowboy_window : public Cowboy, public Window
{
  // ...
};
```

Cowboy_window представляет анимацию ковбоя, и через этот класс пользователь осуществляет управление ковбоем. Мы предпочли множественное наследование агрегации членов с типами ***Cowboy*** и ***Window***, поскольку имеется множество вспомогательных сервисов, определенных как для ***Window***, так и для ***Cowboy***. Хотелось бы передавать таким функциям ***Cowboy_window***, не требуя от программиста никаких дополнительных действий. Это, однако, приводит к проблеме определения в классе ***Cowboy_window*** версий функций ***Cowboy::draw()*** и ***Window::draw()***.

В ***Cowboy_window*** может быть только одна функция ***draw***(). Но поскольку сервисные функции манипулируют объектами ***Window*** и ***Cowboy***, ничего не зная о ***Cowboy_window***, последний должен заместить ***draw***() и для класса ***Cowboy***, и для класса ***Window***. Замещение их обеих единственной функцией будет неправильным, так как, несмотря на одинаковое название это разные функции. Наконец, хотелось бы получить однозначно различающиеся имена для унаследованных в ***Cowboy_window*** функций ***Cowboy::draw***() и ***Window::draw***().

Для решения проблемы введем по дополнительному классу для ***Cowboy*** и для ***Window***. Эти классы введут два новых имени для функций ***draw***(), а также заместят ***draw***() при наследовании от ***Cowboy*** и ***Window*** так, что вызываться будут функции с новыми именами:

```
class CCowboy: public Cowboy     // интерфейс к Cowboy, переименовывающий draw()
{
public:
  virtual int cow_draw() = 0 ;
  void draw() {cow_draw() ; }    // замещаем Cowboy::draw()
};

class WWindow: public Window     // интерфейс к Window, переименовывающий draw()
{
public:
  virtual int win_draw() = 0;
  void draw() {win_draw() ; }    // замещаем Window::draw()
};
```

Теперь мы можем составить ***Cowboy_window*** из интерфейсных классов ***CCowboy*** и ***WWindow***, и заместить ***cow_draw***() и ***win_draw***(), чтобы добиться желаемого эффекта:

```
class Cowboy_window: public CCowboy, public WWindow
{
  // ...
  void cow_draw() ;
  void win_draw() ;
};
```

Заметим, что проблема оказалась серьезной по той причине, что две функции с одинаковыми именами имели аргументы одинакового типа. Если бы их аргументы различались, обычные правила перегрузки гарантировали бы различение функций с одинаковыми именами.

Для каждого способа применения интерфейсных классов можно было бы, в принципе, представить некоторое расширение языка, которое выполняло бы необходимое приспособление чуть более элегантно и эффективно. Однако каждый вариант применения интерфейсных классов встречается не столь уж и часто, так что сопоставление каждому из них дополнительных языковых конструкций внесло бы в язык ненужную сложность. Кроме того, столкновение имен при слиянии двух классовых иерархий тоже нельзя назвать обычным явлением (по сравнению, например, с тем, как часто программист пишет классы), и, к тому же, оно имеет место лишь при объединении иерархий из разных предметных областей — таких как игры и системы оконного интерфейса. Слияние непохожих друг на друга иерархий — во-

обще столь непростая задача, что разрешение конфликтов имен далеко не единственная возникающая при этом проблема; есть еще разные способы инициализации, разные стратегии обработки ошибок и управления памятью и т.д. Мы здесь обсуждаем разрешение конфликтов имен лишь потому, что рассмотренный прием с введением интерфейсных классов, имеющих функции с переадресацией вызовов (forwarding functions), сам по себе интересен и имеет множество иных применений. Его можно применять не только для изменения имен функций, но и для изменения типов их аргументов и возврата, для введения проверок на стадии выполнения и др.

Поскольку функции *CCowboy::draw()* и *WWindow::draw()* переадресуют свой вызов виртуальным функциям, их нельзя оптимизировать встраиванием. Но если компилятор распознает их в качестве переадресующих функций, он сможет оптимизировать код, убрав их из цепочки вызовов.

25.6.1. Подгонка интерфейсов

Основное применение интерфейсных функций состоит в подгонке интерфейса под ожидания пользователей так, чтобы сконцентрировать в рамках интерфейса весь код, который иначе был бы разбросан по всей программе. Например, отсчет индексов в стандартном контейнере *vector* начинается с нуля. Если пользователи не хотят, чтобы индексы занимали диапазон от *0* до *size-1*, они должны приспособить вектора для своих нужд. Например:

```
void f()
{
  vector v<int> (10) ;              // диапазон [0:9]

  // делаем вид, что диапазон [1:10]:
  for (int i = 1; i<=10; i++)
  {
    v[i-1] = 7;                     // подправляем индекс
    // ...
  }
}
```

Еще лучше снабдить вектор произвольными границами:

```
class Vector: public vector<int>
{
  int lb;

public:
  Vector (int low, int high) : vector<int> (high-low+1) {lb=low; }

  int& operator[] (int i) {return vector<int>::operator[] (i-lb) ; }

  int low() {return lb; }
  int high() {return lb+size()-1; }
};
```

Класс *Vector* можно использовать следующим образом:

```
void g()
{
  Vector v(1,10) ;                  // диапазон [1:10]
```

```
  for (int i = 1; i<=10; i++)
  {
    v[i] = 7;
    //...
  }
}
```

Это не приводит к дополнительным по сравнению с предыдущим примером затратам. В то же время, вариант с ***Vector*** намного нагляднее, его легче и читать, и писать, так что он менее подвержен ошибкам.

Интерфейсные классы обычно невелики и (по определению) мало что делают. Однако они становятся необходимы, когда нужно объединить программные компоненты, написанные согласно разным традициям. Например, интерфейсные классы часто используются для предоставления C++-интерфейса к коду, написанному на других языках программирования, а также для изоляции пользовательского кода от деталей библиотеки (оставляя, тем самым, возможность замены библиотеки).

Другое важное применение интерфейсных классов — предоставление проверяемых или ограничительных интерфейсов. Например, нередко требуется обеспечить ограничение целых значений заданным диапазоном. Это можно гарантировать (во время выполнения программы) с помощью простого шаблона:

```
template<int low, int high> class Range
{
  int val;

public:
  class Error { };              // класс исключений

  Range (int i) {Assert<Error> (low<=i&&i<high) ; val = i; }     // см. §24.3.7.2

  Range operator= (int i) {return *this=Range (i) ; }
  operator int () {return val; }
  // ...
};

void f(Range<2, 17>) ;
void g (Range<-10, 10>) ;

void h (int x)
{
  Range<0, 2001> i = x;  // может сгенерировать Range::Error
  int i1 = i;

  f(3) ;
  f(17) ;                // генерируется Range::Error
  g (-7) ;
  g (100) ;              // генерируется Range::Error
}
```

Шаблон ***Range*** легко расширяется для работы с диапазонами произвольных скалярных типов (§25.10[7]).

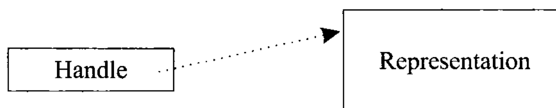
Часто интерфейсный класс, контролирующий доступ к другому классу или выполняющий подгонку его интерфейса, называют *оберткой* (*wrapper*).

25.7. Дескрипторные классы (handles)

Абстрактный класс помогает эффективно разделить интерфейс и его реализации. Однако если следовать §25.3, то получается неразрывная связь между интерфейсом, предоставляемым абстрактным типом, и его реализацией, обеспечиваемой конкретным типом. Например, невозможно отвязать абстрактный итератор от одного источника (скажем, множества) и привязать его к другому (например, потоку) по мере исчерпания первого источника.

Кроме того, если манипулировать объектами, реализующими абстрактный класс, не через указатели или ссылки, то теряются выгоды, предоставляемые виртуальными функциями. Пользовательский код становится зависимым от деталей класса реализации, ибо невозможно выделить память под эти объекты статически или в стеке, не зная точного размера объектов. Использование же указателей или ссылок означает, что бремя управления памятью ложится на пользователя.

Распространенный способ решения этих проблем состоит в разделении одного объекта на два разных: *дескриптора* (*handle*), обеспечивающего пользовательский интерфейс, и *представления* (*representation*), содержащего все или большую часть состояния объекта. Соединяет их указательное поле в дескрипторе. В типичном случае класс дескриптора содержит чуть больше данных, чем один лишь указатель на представление, но не намного. Это означает, что расположение в памяти объектов дескрипторного класса остается неизменным, даже если представление изменяется, и что дескрипторные объекты достаточно малы, чтобы их можно было безболезненно передавать в функции по значению, так что отпадает нужда в указателях или ссылках.



Дескрипторы иллюстрируются классом ***String*** из §11.12; дескрипторный класс обеспечивает интерфейс, контроль за доступом и управления памятью для представления. В данном случае и дескриптор, и представление задаются конкретными классами, но часто представлению соответствуют абстрактные классы.

Рассмотрим абстрактный тип ***Set*** (множество) из §25.3. Как можно ввести для него дескриптор и какие выгоды это даст (и какой ценой)? Отталкиваясь от класса ***Set***, можно определить дескрипторный класс, просто перегрузив операцию ->:

```
template<class T> class Set_handle
{
  Set<T>* rep;

public:
  Set<T>* operator->() {return rep; }
  Set_handle(Set<T>* pp) : rep(pp) { }
};
```

Это не меняет существенно то, как используются множества; просто всюду вместо ***Set&*** или ***Set**** передаются ***Set_handle***. Например:

```
void f(Set_handle<int> s)
{
 for (int* p = s->first() ; p; p = s->next())
 {
  // ...
 }
}

void user()
{
 Set_handle<int> sl(new List_set<int>) ;
 Set_handle<int> v(new Vector_set<int>(100)) ;

 f(sl) ;
 f(v) ;
}
```

Часто требуется, чтобы дескриптор не ограничивался лишь предоставлением доступа. Например, если бы класс *Set* и дескриптор *Set_handle* проектировались совместно, то можно было бы легко обеспечить подсчет ссылок, включив счетчики в объекты *Set*. Но в общем случае, нам не хотелось бы проектировать дескрипторы совместно с управляемыми ими объектами, чтобы не хранить совместно используемые данные в отдельных объектах. Иными словами, помимо интрузивных дескрипторов хотелось бы иметь и неинтрузивные. Вот пример дескрипторного класса, удаляющего объект по исчерпанию ссылок на него:

```
template<class X> class Handle
{
 X* rep;
 int* pcount;

public:
 X* operator->() {return rep; }

 Handle(X* pp) : rep(pp), pcount(new int(1)) { }
 Handle(const Handle& r) : rep(r.rep), pcount(r.pcount) { (*pcount)++; }

 Handle& operator=(const Handle& r)
 {
  if(rep == r.rep) return *this;
  if(--(*pcount) == 0)
  {
   delete rep;
   delete pcount;
  }

  rep = r.rep;
  pcount = r.pcount;
  (*pcount)++;
  return *this;
 }

 ~Handle() { if(--(*pcount) == 0) {delete rep; delete pcount; } }
 // ...
};
```

Такие дескрипторы можно свободно передавать:

```
void f1(Handle<Set>);

Handle<Set> f2()
{
  Handle<Set> h(new List_set<int>);
  // ...
return h;
}

void g()
{
  Handle<Set> hh=f2();
  f1(hh);
  // ...
}
```

Множество, созданное в *f2*(), будет автоматически удалено на выходе из функции *g*(), если только *f1*() не создает ссылку на него (программисту здесь этого даже знать не надо).

Естественно, такое удобство не дается даром, но для большинства прикладных программ цена хранения и обслуживания счетчика ссылок вполне приемлема.

Иногда нужно извлечь указатель из дескриптора и использовать его напрямую. Например, если нужно передать указатель функции, которая ничего не знает о дескрипторе. Это неплохо работает, если, конечно, функция не уничтожит переданный ей объект. Также может быть полезной операция привязки дескриптора к новому представлению:

```
template<class X> class Handle
{
  // ...
  X* get_rep() {return rep; }

  void bind(X* pp)
{
    if(pp != rep)
    {
     if(--(*pcount) == 0)
      {
delete rep;
        *pcount = 1;            // обновить pcount
      }
else
        pcount = new int(1);    // новый pcount

     rep = pp;
    }
}
};
```

Заметим, что наследование новых классов от ***Handle*** не особенно полезно. Это конкретный тип без виртуальных функций. Идея состоит в том, чтобы иметь один дескрипторный класс для семейства классов, определяемых одним базовым клас-

сом. Наследование от этого базового класса является мощной технологией. Она применима и к узловым классам, и к абстрактным типам.

Как уже сказано, ***Handle*** не предназначен для порождения производных классов. Чтобы получить класс, действующий как истинный указатель с подсчетом ссылок, нужно ***Handle*** скомбинировать с ***Ptr*** из §13.6.3.1 (см. §25.10[2]).

Дескриптор, предоставляющий интерфейс, почти идентичный классу, для которого он обеспечивает доступ, часто называют *заместителем* (*proxy*). Это типично для дескрипторов, ссылающихся на удаленные объекты (объекты на удаленных компьютерах).

25.7.1. Операции в дескрипторных классах

Перегрузка операции -> позволяет дескриптору получать управление и выполнять некоторую работу при каждом обращении к объекту. Например, можно собрать статистику о числе обращений к объекту через дескриптор:

```
template <class T> class Xhandle
{
  T* rep;
  int no_of_accesses;

public:
  T* operator->() {no_of_accesses++; return rep; }
  // ...
};
```

Дескрипторы, которым нужно выполнить работу и *до*, и *после* обращения, требуют более изощренного программирования. Рассмотрим, к примеру, множество, для которого нужно выполнять блокировку при вставках или удалениях элементов. Существенно, что интерфейс класса представления должен быть повторен в дескрипторном классе:

```
template<class T> class Set_controller
{
  Set<T>* rep;
  Lock lock;
  // ...

public:
  void insert(T* p) {Lock_ptr x(lock) ; rep->insert(p) ; }        // см. §14.4.1
  void remove(T* p) {Lock_ptr x(lock) ; rep->remove(p) ; }

  int is_member(T* p) {return rep->is_member(p) ; }

  T get_first() {T* p = rep->first() ; return p ? *p : T() ; }
  T get_next() {T* p = rep->next() ; return p ? *p : T() ; }

  T first() {Lock_ptr x(lock) ; T tmp = *rep->first() ; return tmp; }
  T next() {Lock_ptr x(lock) ; T tmp = *rep->next() ; return tmp; }
  // ...
};
```

Введение всех этих переадресующих функций довольно утомительно, но не слишком сложно и не очень дорого с точки зрения влияния на производительность программы.

Отметим, что далеко не все функции *Set* нуждаются в блокировке. По моему опыту, это типично для большинства программ, нуждающихся в пред- и пост-действиях. Блокировка абсолютно всех действий в некоторых системных мониторах приводит к ухудшению параллельного исполнения.

Важной причиной для тщательного определения операций дескриптора при перегрузке в нем операции -> является возможность наследования от *Set_controller*. Однако многие выгоды от дескрипторов становятся сомнительными, когда в производный класс добавляются поля данных. В частности, объем контролируемых дескриптором общих данных снижается относительно объема данных, сосредоточенных в каждом дескрипторном объекте.

25.8. Прикладные среды разработки (application frameworks)

Компоненты, построенные из классов, рассмотренных в §25.2 — §25.7, поддерживают проектирование и повторное использование кода, предоставляя как строительные блоки, так и способы их комбинирования; прикладной программист строит каркас программы, в который эти общеупотребительные строительные блоки включаются. Альтернативным (и часто более амбициозным) подходом к поддержке проектирования и повторного использования является автоматическое построение *каркаса приложения* в рамках *прикладных сред разработки* (*application frameworks*), когда программист лишь добавляет к каркасу приложения специфические блоки собственного изготовления. Классы, из которых автоматически строится общий каркас приложения, обладают столь «жирными» интерфейсами, что вряд ли являются классами в обычном понимании. Они приближают построение законченной программы, если не считать того, что они ничего специфического (характерного для нового приложения) не делают. Все конкретные действия предоставляются непосредственно прикладным программистом.

В качестве примера рассмотрим фильтр, то есть программу, которая считывает поток ввода, выполняет какие-то действия с введенными данными, выводит что-то в поток вывода и (возможно) выдает некоторый окончательный результат. Простая среда разработки таких программ обеспечивает набор операций, которые могут потребоваться прикладному программисту:

```
class Filter
{
public:
  class Retry
  {
  public:
    virtual const char* message() {return 0; }
  };

  virtual void start() { }
  virtual int read() = 0;
  virtual void write() { }
  virtual void compute() { }
  virtual int result() = 0;
```

```
  virtual int retry (Retry& m) {cerr << m.message() << '\n'; return 2;}
  virtual ~Filter() { }
};
```

Функции, которые должен определять производный класс, объявлены чисто виртуальными; остальные функции определены таким образом, что просто ничего не делают.

Среда разработки также обеспечивает главный цикл и рудиментарный механизм обработки ошибок:

```
int main_loop (Filter* p)
{
 for (;;)
 {
  try
  {
   p->start();
   while (p->read())
   {
    p->compute();
    p->write();
   }
   return p->result();
  }
  catch (Filter::Retry& m)
  {
   if (int i = p->retry(m)) return i;
  }
  catch (...)
  {
   cerr << "Fatal filter error\n";
   return 1;
  }
 }
}
```

Я (в качестве прикладного программиста) свою собственную часть программы мог бы написать следующим образом:

```
class My_filter: public Filter
{
 istream& is;
 ostream& os;
 int nchar;

public:
 int read() {char c; is.get(c); return is.good();}
 void compute() {nchar++;}
 int result() {os << nchar << " characters read\n"; return 0;}

 My_filter(istream& ii, ostream& oo) : is(ii), os(oo), nchar(0) { }
};
```

и запустить все это на выполнение:

```
int main ()
{
  My_filter f(cin.cout) ;
return main_loop (&f) ;
}
```

Естественно, чтобы быть по-настоящему полезной, среда разработки должна предоставлять больше структурных возможностей и намного больше сервисов, чем наш простой пример. В частности, среды разработки обычно являются иерархиями узловых классов. Благодаря тому, что прикладному программисту остается добавлять «листовые» классы (классы-листья), которые прорастают глубоко внутри иерархии, достигается изрядная общность прикладных программ между собой и высокая степень повторного использования служебных сервисов среды разработки. Среды разработки обычно дополняются библиотеками классов, которые нужны прикладному программисту при написании специфического кода «работающих» классов.

25.9. Советы

1. Делайте осознанные решения по способам использования классов (осознанные как проектировщиком, так и пользователем); §25.1.
2. Учитывайте достоинства и недостатки применения разных видов классов; §25.1.
3. Используйте конкретные типы для представления простых независимых концепций; §25.2.
4. Используйте конкретные типы для представления концепций, для которых важна близкая к оптимальной эффективность; §25.2.
5. Не наследуйте от конкретного класса; §25.2.
6. Используйте абстрактные классы для представления интерфейсов в случаях, когда реализации объектов могут варьироваться; §25.3.
7. Используйте абстрактные классы для представления интерфейсов в случаях, когда разные реализации объектов должны сосуществовать; §25.3.
8. Используйте абстрактные классы для предоставления новых интерфейсов к уже существующим типам; §25.3.
9. Используйте узловые классы там, где схожие концепции разделяют большую часть реализации; §25.4.
10. Используйте узловые классы для постепенного расширения реализации; §25.4.
11. Используйте RTTI для выявления интерфейса объекта; §25.4.1.
12. Используйте классы для ассоциации действий с состоянием (данными); §25.5.
13. Используйте классы для представления действий (операций), которые нужно сохранять, передавать или откладывать их выполнение; §25.5.

14. Используйте интерфейсные классы для адаптирования классов к новым способам их использования (без модификации самих классов); §25.6.
15. Используйте интерфейсные классы, чтобы добавить проверки; §25.6.1.
16. Применяйте дескрипторы, чтобы избежать непосредственного использования указателей или ссылок; §25.7.
17. Применяйте дескрипторы для управления разделяемыми представлениями; §25.7.
18. Применяйте прикладные среды разработок в тех областях, где особенности программ допускают их стандартную структуру; §25.8.

25.10. Упражнения

1. (*1) Шаблон ***Io*** из §25.4.1 не работает для встроенных типов. Исправьте эту ситуацию.
2. (*1.5) Шаблон ***Handle*** из §25.7 не отражает отношений наследования между классами, которыми он управляет. Сделайте так, чтобы он отражал эти зависимости. То есть чтобы можно было присваивать ***Handle<Circle>*** объектам типа ***Handle<Shape>***, но не наоборот.
3. (*2.5) Отталкиваясь от класса ***String***, определите другой класс, реализующий представление строк с помощью виртуальных функций. Сравните производительность двух классов. Постарайтесь определить разумный класс, который оптимально реализовывался бы в форме открытого наследования от строкового класса с виртуальными функциями.
4. (*4) Изучите пару широко используемых библиотек. Классифицируйте их классы в терминах конкретных типов, абстрактных типов, узловых классов, дескрипторных классов и интерфейсных классов. Используются ли в этих библиотеках абстрактные узловые и конкретные узловые классы? Можно ли классы этих библиотек классифицировать как-то иначе? Используются ли жирные интерфейсы? Какие средства применяются для идентификации типов на стадии выполнения? Какова стратегия управления памятью?
5. (*2) Воспользуйтесь средой разработки ***Filter*** (§25.8) для реализации программы, которая устраняет смежные дублированные слова в потоке ввода, и перенаправляет исправленный поток слов в поток вывода.
6. (*2) Воспользуйтесь средой разработки ***Filter*** для реализации программы, которая подсчитывает частоту слов в потоке ввода и помещает в поток вывода пары (слово, частота) в порядке возрастания частоты.
7. (*1.5) Напишите шаблон ***Range***, который принимает тип и диапазон значений элементов в качестве параметров шаблона.
8. (*1) Напишите шаблон ***Range***, который принимает диапазон значений в качестве параметров конструктора.
9. (*2) Напишите простой строковый класс, который не обрабатывает никаких ошибок. Напишите другой класс, который проверяет доступ к первому. Обсудите все «за» и «против» разделения базовых функций и проверки ошибок.

10. (*2.5) Реализуйте систем

у объектного ввода/вывода из §25.4.1 для нескольких типов, включая целые числа, строки и классовую иерархию по вашему выбору.

11. (*2.5) Определите класс ***Storable*** в качестве абстрактного базового класса с виртуальными функциями ***write_out***() и ***read_in***(). Для простоты предположите, что символьная строка описывает место для долговременного хранилища. Воспользуйтесь классом ***Storable***, чтобы написать средство, позволяющее сохранять на диске и читать с него объекты классов, производных от ***Storable***. Протестируйте это средство с парой классов по вашему выбору.

12. (*4) Разработайте базовый класс ***Persistent*** с операциями ***save***() и ***no_save***(), которые определяют, записываются ли объекты в долговременное хранилище деструктором. Кроме указанных операций какие еще полезные операции мог бы предоставлять класс ***Persistent***? Протестируйте класс ***Persistent*** с парой классов по вашему выбору. Является ли ***Persistent*** узловым классом, конкретным типом или абстрактным типом? Почему?

13. (*3) Напишите класс ***Stack***, для которого можно сменить представление на стадии выполнения программы. Подсказка: «любую проблему можно решить, введя еще один уровень косвенности».

14. (*3.5) Определите класс ***Oper***, содержащий идентификатор типа ***Id*** (***string*** или С-строка) и операцию (указатель на функцию или функциональный объект). Определите класс ***Cat_object***, который содержит список элементов типа ***Oper*** и ***void****. Снабдите ***Cat_object*** операциями ***add_oper***(***Oper***), которая добавляет ***Oper*** к списку; ***remove_oper***(***Id***), которая удаляет ***Oper***, идентифицируемую с помощью ***Id*** из списка; ***operator***() (***Id***, ***arg***), которая активизирует ***Oper***, идентифицируемую с помощью ***Id***. Реализуйте стек котов посредством ***Cat_object***. Напишите небольшую программу для тестирования этих классов.

15. (*3) Определите шаблон ***Object*** на базе класса ***Cat_object***. Используйте ***Object*** для реализации стека строк. Напишите небольшую программу для тестирования этого шаблона.

16. (*2.5) Определите вариант класса ***Object*** под именем ***Class***, который гарантирует, что объекты с одинаковыми операциями совместно разделяют список операций. Напишите небольшую программу для тестирования этого шаблона.

17. (*2) Определите шаблон ***Stack***, который предоставляет традиционный и безопасный по типу интерфейс к стеку, реализованному шаблоном ***Object***. Сравните этот стек с вариантами стековых классов из предыдущих упражнений. Напишите небольшую программу для тестирования этого шаблона.

18. (*3) Напишите класс для представления операций, которые нужно отправлять для исполнения на удаленный компьютер. Протестируйте этот класс, либо отправляя команды на другой компьютер, либо при помощи записи команд в файл с последующим их чтением и исполнением.

19. (*2) Напишите класс для композиции операций, представленных объектами функциональных классов (объектами-функциями). Для двух объектов ***f*** и ***g***, ***Compose***(***f***, ***g***) должно порождать объект, который может быть активизирован с аргументом ***x***, пригодным для ***g***, и с возвратом ***f***(***g***(***x***)), при условии, что возврат ***g***() является приемлемым аргументом для ***f***().

Приложения и предметный указатель

В приложениях представлены грамматика языка C++, обсуждение вопросов совместимости C и C++ и вопросов совместимости между стандартом C++ и предстандартными версиями, а также множество технических деталей. Для стандартной библиотеки представлены средства интернационализации, а также концепции, гарантии и методы реализации безопасности при генерации исключений. Предметный указатель достаточно обширен и является неотъемлемой частью данной книги.

Главы

A. Грамматика

B. Совместимость

C. Технические подробности

D. Локализация

E. Исключения и безопасность стандартной библиотеки

Предметный указатель

Приложение А

Грамматика

Самая большая опасность для преподавателя —
учить словам, а не вещам.
— Марк Блок

Введение — ключевые слова — лексические соглашения — программы — выражения — операторы — объявления — классы — производные классы — особые функции-члены — перегрузка — шаблоны — обработка исключений — директивы препроцессора.

А.1. Введение

Итоговое концентрированное изложение синтаксиса C++ призвано быть формальным дополнением и должно способствовать лучшему его пониманию. Оно не является абсолютно точным и строгим описанием языка C++. В частности, описанная здесь грамматика содержит некоторые расширения допустимых для C++ конструкций. Требуется применять правила разрешения неоднозначностей для различения выражений от объявлений (§А.5, §А.7). Кроме того, для выправления синтаксически корректных, но бессмысленных конструкций нужно применять правило доступа, правило типов и правило разрешения неоднозначностей.

Стандартные грамматики C и C++ различаются с синтаксической точки зрения минимально, в большей степени через ограничения. Это способствует точности, но не читаемости кода.

А.2. Ключевые слова

Новые контекстнозависимые ключевые слова вводятся в программу с помощью **typedef** (§4.9.7), пространств имен (§8.2), с помощью объявлений классов (глава 10), перечислений (§4.8) и шаблонов (глава 13)[1]:

typedef-name:
  identifier

namespace-name:
  original-namespace-name
  namespace-alias

original-namespace-name:
  identifier

namespace-alias:
  identifier

class-name:
  identifier
  template-id

enum-name:
  identifier

template-name:
  identifier

Заметим, что *typedef-имя*, именующее класс, есть также и *class-имя*.

Если идентификатор явным образом не именует тип, то считается, что он именует что угодно, но только не тип (см. также §С.13.5).

Ниже перечислены ключевые слова языка С++:

Ключевые слова С++				
and	***continue***	***goto***	***public***	***try***
and_eq	***default***	***if***	***register***	***typedef***
asm	***delete***	***inline***	***reinterpret_cast***	***typeid***
auto	***do***	***int***	***return***	***typename***
bitand	***double***	***long***	***short***	***union***
bitor	***dynamic_cast***	***mutable***	***signed***	***unsigned***
bool	***else***	***namespace***	***sizeof***	***using***
break	***enum***	***new***	***static***	***virtual***
case	***explicit***	***not***	***static_cast***	***void***

[1] Ввиду большой практической ценности и фактической самоочевидности (для разработчиков компиляторов) представляем авторское формализованное изложение грамматики в точном оригинальном виде (без перевода простейших терминов). Здесь $_{opt}$ — это сокращение от optional (необязательно). — *Прим. ред.*

Ключевые слова C++				
catch	***export***	***not_eq***	***struct***	***volatile***
char	***extern***	***operator***	***switch***	***wchar_t***
class	***false***	***or***	***template***	***while***
compl	***float***	***or_eq***	***this***	***xor***
const	***for***	***private***	***throw***	***xor_eq***
const_cast	***friend***	***protected***	***true***	

А.3. Лексические соглашения

Стандартные грамматики языков C и C++ представляют лексические соглашения как грамматические сущности. Это добавляет точности, но не всегда способствует читабельности текстов:

hex-quad:
  hexadecimal-digit hexadecimal-digit hexadecimal-digit hexadecimal-digit

universal-character-name:
  `\u` *hex-quad*
  `\U` *hex-quad hex-quad*

preprocessing-token:
  header-name
  identifier
  pp-number
  character-literal
  string-literal
  preprocessing-op-or-punc
  each non-white-space character that cannot be one of the above

token:
  identifier
  keyword
  literal
  operator
  punctuator

header-name:
  <h-char-sequence>
  "q-char-sequence"

h-char-sequence:
  h-char
  h-char-sequence h-char

h-char:
  any member of the source character set except new-line and >

q-char-sequence:
  q-char
  q-char-sequence q-char

q-char:
any member of the source character set except new-line and "

pp-number:
digit
. digit
pp-number digit
pp-number nondigit
pp-number e sign
pp-number E sign
pp-number .

identifier:
nondigit
identifier nondigit
identifier digit

nondigit: one of
universal-character-name

```
_ a b c d e f g h i j k l m n o p q r s t u v w x y z
      A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
```

digit: one of

```
0 1 2 3 4 5 6 7 8 9
```

preprocessing-op-or-punc: one of

```
{      }      [      ]      #      ##     (      )      <:     :>     <%     %>     %:%:
%:     ;      :      ?      ::     .      .*     +      -      *      /      %      ^
&      |      ~      !      =      <      >      +=     -=     *=     /=     %=     ^=
&=     |=     <<=    >>=    <<     >>     ==     !=     <=     >=     &&     ||     ++
--     ,      ->     ->*    ...    new    delete        and    and_eq        bitand
bitor         comp          not    or     not_eq        xor    or_eq         xor_eq
```

literal:
integer-literal
character-literal
floating-literal
string-literal
boolean-literal

integer-literal:
decimal-literal integer-suffix$_{opt}$
octal-literal integer-suffix$_{opt}$
hexadecimal-literal integer-suffix$_{opt}$

decimal-literal:
nonzero-digit
decimal-literal digit

octal-literal:
0
octal-literal octal-digit

hexadecimal-literal:
0x *hexadecimal-digit*
0X *hexadecimal-digit*

hexadecimal-literal hexadecimal-digit

nonzero-digit: one of

`1 2 3 4 5 6 7 8 9`

octal-digit: one of

`0 1 2 3 4 5 6 7`

hexadecimal-digit: one of

`0 1 2 3 4 5 6 7 8 9`
`a b c d e f`
`A B C D E F`

integer-suffix:
unsigned-suffix long-suffix$_{opt}$
long-suffix unsigned-suffix$_{opt}$

unsigned-suffix: one of

`u U`

long-suffix: one of

`l L`

character-literal:
'c-char-sequence'
`L`*'c-char-sequence'*

c-char-sequence:
c-char
c-char-sequence c-char

c-char:
any member of the source character set except the single-quote, backslash, or new-line character
escape-sequence
universal-character-name

escape-sequence:
simple-escape-sequence
octal-escape-sequence
hexadecimal-escape-sequence

simple-escape-sequence: one of

`\' \" \? \\ \a \b \f \n \r \t \v`

octal-escape-sequence:
`\` *octal-digit*
`\` *octal-digit octal-digit*
`\` *octal-digit octal-digit octal-digit*

hexadecimal-escape-sequence:
`\x` *hexadecimal-digit*
hexadecimal-escape-sequence hexadecimal-digit

floating-literal:
fractional-constant exponent-part$_{opt}$ *floating-suffix*$_{opt}$
digit-sequence exponent-part floating-suffix$_{opt}$

fractional-constant:
digit-sequence$_{opt}$. *digit-sequence*
digit-sequence .

exponent-part:
e *sign*$_{opt}$ *digit-sequence*
E *sign*$_{opt}$ *digit-sequence*

sign: one of
\+ -

digit-sequence:
digit
digit-sequence digit

floating-suffix: one of
f l F L

string-literal:
"*s-char-sequence*$_{opt}$"
L"*s-char-sequence*$_{opt}$"

s-char-sequence:
s-char
s-char-sequence s-char

s-char:
any member of the source character set except double-quote, backslash, or new-line
escape-sequence
universal-character-name

boolean-literal:
false
true

При составлении лексемы выбирается *наиболее длинная возможная последовательность символов.* Например, ***double*** составляет единую лексему, а не комбинацию из ***do*** и ***uble***; последовательность символов +++ есть ++ с последующим +.

А.4. Программы

Программа — это набор единиц трансляции, объединяемых в процессе компоновки (linking) (§9.4). *Единица трансляции* (*translation-unit*), часто называемая *исходным файлом* (*source file*) — это последовательность *объявлений* (*declarations*):

translation-unit:
declaration-seq$_{opt}$

А.5. Выражения

Выражения рассматриваются в главе 6 с подробным перечислением в §6.2. Определение *списка-выражений* (*expression-list*) эквивалентно определению *выражения* (*expression*). Имеются два правила для отличия запятой как операции (операция следования — sequencing) и как разделителя списка аргументов функции (§6.2.2).

primary-expression:
  literal
  `this`
  `::` *identifier*
  `::` *operator-function-id*
  `::` *qualified-id*
  `(` *expression* `)`
  id-expression

id-expression:
  unqualified-id
  qualified-id

unqualified-id:
  identifier
  operator-function-id
  conversion-function-id
  ~ *class-name*
  template-id

qualified-id:
  nested-name-specifier `template`$_{opt}$ *unqualified-id*

nested-name-specifier:
  class-or-namespace-name `::` *nested-name-specifier*$_{opt}$
  class-or-namespace-name `::` `template` *nested-name-specifier*

class-or-namespace-name:
  class-name
  namespace-name

postfix-expression:
  primary-expression
  postfix-expression `[` *expression* `]`
  postfix-expression `(` *expression-list*$_{opt}$ `)`
  simple-type-specifier `(` *expression-list*$_{opt}$ `)`
  `typename` `::`$_{opt}$*nested-name-specifier identifier* `(` *expression-list*$_{opt}$ `)`
  `typename` `::`$_{opt}$*nested-name-specifier* `template`$_{opt}$ *template-id* `(` *expression-list*$_{opt}$ `)`
  postfix-expression `.` `template`$_{opt}$ `::`$_{opt}$ *id-expression*
  postfix-expression `->` *template*$_{opt}$ `::`$_{opt}$ *id-expression*
  postfix-expression `.` *pseudo-destructor-name*
  postfix-expression `->` *pseudo-destructor-name*
  postfix-expression `++`
  postfix-expression `--`
  `dynamic_cast` < *type-id* > `(` *expression* `)`
  `static_cast` < *type-id* > `(` *expression* `)`
  `reinterpret_cast` < *type-id* > `(` *expression* `)`
  `const_cast` <*type-id* > `(` *expression* `)`
  `typeid` `(` *expression* `)`
  `typeid` `(` *type-id* `)`

expression-list:
  assignment-expression
  expression-list , *assignment-expression*

pseudo-destructor-name:
`::`$_{opt}$ *nested-name-specifier*$_{opt}$ *type-name* `::` `~` *type-name*
`::`$_{opt}$ *nested-name-specifier* `template` *template-id* `::` `~` *type-name*
`::`$_{opt}$ *nested-name-specifier*$_{opt}$ `~` *type-name*

unary-expression:
postfix-expression
`++` *cast-expression*
`--` *cast-expression*
unary-operator cast-expression
`sizeof` *unary-expression*
`sizeof` `(` *type-id* `)`
new-expression
delete-expression

unary-operator: one of
`*` `&` `+` `-` `!` `~`

new-expression:
`::`$_{opt}$ `new` *new-placement*$_{opt}$ *new-type-id new-initializer*$_{opt}$
`::`$_{opt}$ `new` *new-placement*$_{opt}$ `(` *type-id* `)` *new-initializer*$_{opt}$

new-placement:
`(` *expression-list* `)`

new-type-id:
type-specifier-seq new-declarator$_{opt}$

new-declarator:
ptr-operator new-declarator$_{opt}$
direct-new-declarator

direct-new-declarator:
`[` *expression* `]`
direct-new-declarator `[` *constant-expression* `]`

new-initializer:
`(` *expression-list*$_{opt}$ `)`

delete-expression:
`::`$_{opt}$ `delete` *cast-expression*
`::`$_{opt}$ `delete` `[ ]` *cast-expression*

cast-expression:
unary-expression
`(` *type-id* `)` *cast-expression*

pm-expression:
cast-expression
pm-expression `.*` *cast-expression*
pm-expression `->*` *cast-expression*

multiplicative-expression:
pm-expression
multiplicative-expression `*` *pm-expression*
multiplicative-expression `/` *pm-expression*
multiplicative-expression `%` *pm-expression*

additive-expression:
multiplicative-expression
additive-expression + *multiplicative-expression*
additive-expression - *multiplicative-expression*

shift-expression:
additive-expression
shift-expression << *additive-expression*
shift-expression >> *additive-expression*

relational-expression:
shift-expression
relational-expression < *shift-expression*
relational-expression > *shift-expression*
relational-expression <= *shift-expression*
relational-expression >= *shift-expression*

equality-expression:
relational-expression
equality-expression == *relational-expression*
equality-expression != *relational-expression*

and-expression:
equality-expression
and-expression & *equality-expression*

exclusive-or-expression:
and-expression
exclusive-or-expression ^ *and-expression*

inclusive-or-expression:
exclusive-or-expression
inclusive-or-expression | *exclusive-or-expression*

logical-and-expression:
inclusive-or-expression
logical-and-expression && *inclusive-or-expression*

logical-or-expression:
logical-and-expression
logical-or-expression || *logical-and-expression*

conditional-expression:
logical-or-expression
logical-or-expression ? *expression* : *assignment-expression*

assignment-expression:
conditional-expression
logical-or-expression assignment-operator assignment-expression
throw-expression

assignment-operator: one of
= *= /= %= += -= >>= <<= &= ^= |=

expression:
assignment-expression
expression , *assignment-expression*

constant-expression:
conditional-expression

Грамматические неоднозначности возникают из-за идентичности приведения типа в функциональном стиле и объявлений. Например:

```
int x;
void f()
{
  char(x);          // приведение типа x к char или объявление переменной x типа char?
}
```

Все такие неоднозначности разрешаются в пользу объявлений (если нечто выглядит как объявление, это и есть объявление). Например:

```
T(a)->m;            // оператор-выражение
T(a)++;             // оператор-выражение

T(*e)(int(3));      // объявление
T(f)[4];            // объявление

T(a);               // объявление
T(a)=m;             // объявление
T(*b)();            // объявление
T(x),y,z=7;         // объявление
```

Единственная информация, связанная с именем, это является ли оно именем типа или шаблона. Если это нельзя однозначно установить, то имя считается принадлежащим чему угодно, но только не типу (и не шаблону).

Конструкция ***template*** *неквалифицированный-идентификатор* означает, что *неквалифицированный-идентификатор* относится к имени шаблона (если по контексту это не может быть определено) (см. §С.13.6).

А.6. Операторы

См. §6.3.

statement:
  labeled-statement
  expression-statement
  compound-statement
  selection-statement
  iteration-statement
  jump-statement
  declaration-statement
  try-block

labeled-statement:
  identifier : statement
  `case` *constant-expression : statement*
  `default` *: statement*

expression-statement:
  $expression_{opt}$ *;*

compound-statement:
  { $statement\text{-}seq_{opt}$ }

statement-seq:
    statement
    statement-seq statement

selection-statement:
    `if` (*condition*) *statement*
    `if` (*condition*) *statement* `else` *statement*
    `switch` (*condition*) *statement*

condition:
    expression
    type-specifier-seq declarator = *assignment-expression*

iteration-statement:
    `while` (*condition*) *statement*
    `do` *statement while* (*expression*) ;
    `for` (*for-init-statement condition*$_{opt}$; *expression*$_{opt}$) *statement*

for-init-statement:
    expression-statement
    simple-declaration

jump-statement:
    `break` ;
    `continue` ;
    `return` *expression*$_{opt}$;
    `goto` *identifier* ;

declaration-statement:
    block-declaration

А.7. Объявления

Структура объявлений рассмотрена в главе 4, перечисления — в §4.8, указатели и массивы в главе 5, функции — в главе 7, пространства имен — в §8.2, директивы компоновки — в §9.2.4, классы хранения — в §10.4.

declaration-seq:
    declaration
    declaration-seq declaration

declaration:
    block-declaration
    function-definition
    template-declaration
    explicit-instantiation
    explicit-specialization
    linkage-specification
    namespace-definition

block-declaration:
    simple-declaration
    asm-definition
    namespace-alias-definition

using-declaration
using-directive

simple-declaration:
decl-specifier-seq$_{opt}$ *init-declarator-list*$_{opt}$;

decl-specifier:
storage-class-specifier
type-specifier
function-specifier
`friend`
`typedef`

decl-specifier-seq:
decl-specifier-seq$_{opt}$ *decl-specifier*

storage-class-specifier:
`auto`
`register`
`static`
`extern`
`mutable`

function-specifier:
`inline`
`virtual`
`explicit`

typedef-name:
identifier

type-specifier:
simple-type-specifier
class-specifier
enum-specifier
elaborated-type-specifier
cv-qualifier

simple-type-specifier:
`::`$_{opt}$ *nested-name-specifier*$_{opt}$ *type-name*
`::`$_{opt}$ *nested-name-specifier* `template`$_{opt}$ *template-id*
char
wchar_t
bool
short
int
long
signed
unsigned
float
double
void

type-name:
class-name
enum-name
typedef-name

elaborated-type-specifier:
class-key `::`opt *nested-name-specifier*opt *identifier*
enum `::`opt *nested-name-specifier*opt *identifier*
typename `::`opt *nested-name-specifier identifier*
typename `::`opt *nested-name-specifier* `template`opt *template-id*

enum-name:
identifier

enum-specifier:
`enum` *identifier*opt { *enumerator-list*opt }

enumerator-list:
enumerator-definition
enumerator-list, *enumerator-definition*

enumerator-definition:
enumerator
enumerator = *constant-expression*

enumerator:
identifier

namespace-name:
original-namespace-name
namespace-alias

original-namespace-name:
identifier

namespace-definition:
named-namespace-definition
unnamed-namespace-definition

named-namespace-definition:
original-namespace-definition
extension-namespace-definition

original-namespace-definition:
`namespace` *identifier* { *namespace-body* }

extension-namespace-definition:
`namespace` *original-namespace-name* { *namespace-body* }

unnamed-namespace-definition:
`namespace` { *namespace-body* }

namespace-body:
*declaration-seq*opt

namespace-alias:
identifier

namespace-alias-definition:
  `namespace` *identifier* = *qualified-namespace-specifier* ;

qualified-namespace-specifier:
  `::`$_{opt}$ *nested-name-specifier*$_{opt}$ *namespace-name*

using-declaration:
  `using typename`$_{opt}$ `::`$_{opt}$ *nested-name-specifier unqualified-id* ;
  `using ::` *unqualified-id* ;

using-directive:
  `using namespace ::`$_{opt}$ *nested-name-specifier*$_{opt}$ *namespace-name* ;

asm-definition:
  `asm (` *string-literal* `) ;`

linkage-specification:
  `extern` *string-literal* { *declaration-seq*$_{opt}$ }
  `extern` *string-literal declaration*

Грамматика допускает произвольный уровень вложения объявлений. Однако имеются некоторые семантические ограничения. Например, не допускается вложения функций (определения функции в теле другой функции).

Список спецификаторов, начинающих объявление, не может быть пустым (нет умолчательного ***int***; §B.2) и включает самую длинную из всех возможных последовательность спецификаторов. Например:

```
typedef int I;
void f(unsigned I) {/* ... */}
```

Здесь ***f***`()` принимает неименованный ***unsigned int***.

С помощью ***asm***`()` вводятся фрагменты ассемблерного кода. Более детальный смысл зависит от реализации, но общая идея состоит в том, что строка подается как ассемблерный код, подлежащей внедрению в компилируемый поток в указанном месте.

Модификатор ***register*** означает намек компилятору на то, чтобы он попытался оптимизировать частый доступ к переменной, но это большинство современных компиляторов понимают и без подсказок.

А.7.1. Деклараторы

См. §4.9.1, главу 5 (указатели и массивы), §7.7 (указатели на функции) и §15.5 (указатели на члены классов).

init-declarator-list:
  init-declarator
  init-declarator-list , *init-declarator*

init-declarator:
  declarator initializer$_{opt}$

declarator:
  direct-declarator
  ptr-operator declarator

direct-declarator:
  declarator-id
  direct-declarator (*parameter-declaration-clause*) *cv-qualifier-seq*$_{opt}$
    exception-specification$_{opt}$
  direct-declarator [*constant-expression*$_{opt}$]
  (*declarator*)

ptr-operator:
  * *cv-qualifier-seq*$_{opt}$
  &
  ::$_{opt}$ *nested-name-specifier* * *cv-qualifier-seq*$_{opt}$

cv-qualifier-seq:
  cv-qualifier cv-qualifier-seq$_{opt}$

cv-qualifier:
  `const`
  `volatile`

declarator-id:
  ::$_{opt}$ *id-expression*
  ::$_{opt}$ *nested-name-specifier*$_{opt}$ *type-name*

type-id:
  type-specifier-seq abstract-declarator$_{opt}$

type-specifier-seq:
  type-specifier type-specifier-seq$_{opt}$

abstract-declarator:
  ptr-operator abstract-declarator$_{opt}$
  direct-abstract-declarator

direct-abstract-declarator:
  direct-abstract-declarator$_{opt}$ (*parameter-declaration-clause*) *cv-qualifier-seq*$_{opt}$
    exception-specification$_{opt}$
  direct-abstract-declarator$_{opt}$ [*constant-expression*$_{opt}$]
  (*abstract-declarator*)

parameter-declaration-clause:
  parameter-declaration-list$_{opt}$ ...$_{opt}$
  parameter-declaration-list , ...

parameter-declaration-list:
  parameter-declaration
  parameter-declaration-list , *parameter-declaration*

parameter-declaration:
  decl-specifier-seq declarator
  decl-specifier-seq declarator = *assignment-expression*
  decl-specifier-seq abstract-declarator$_{opt}$
  decl-specifier-seq abstract-declarator$_{opt}$ = *assignment-expression*

function-definition:
  decl-specifier-seq$_{opt}$ *declarator ctor-initializer*$_{opt}$ *function-body*
  decl-specifier-seq$_{opt}$ *declarator function-try-block*

function-body:
  compound-statement

initializer:
  = *initializer-clause*
  (*expression-list*)

initializer-clause:
  assignment-expression
  { *initializer-list* , $_{opt}$ }
  { }

initializer-list:
  initializer-clause
  initializer-list , *initializer-clause*

Спецификатор ***volatile*** — это подсказка компилятору, что объект может изменить свое значение способом, не описанным в языке программирования, так что агрессивной оптимизации следует избегать. Например, часы реального времени могут быть объявлены следующим образом:

extern const volatile long clock;

Два последовательных считывания объекта ***clock*** могут порождать разные результаты.

А.8. Классы

См. главу 10.

class-name:
  identifier
  template-id

class-specifier:
  class-head { *member-specification*$_{opt}$ }

class-head:
  class-key identifier$_{opt}$ *base-clause*$_{opt}$
  class-key nested-name-specifier identifier base-clause$_{opt}$
  class-key nested-name-specifier$_{opt}$ *template-id base-clause*$_{opt}$

class-key:
  `class`
  `struct`
  `union`

member-specification:
  member-declaration member-specification$_{opt}$
  access-specifier : *member-specification*$_{opt}$

member-declaration:
  decl-specifier-seq$_{opt}$ *member-declarator-list*$_{opt}$;
  function-definition ;$_{opt}$
  ::$_{opt}$ *nested-name-specifier* `template`$_{opt}$ *unqualified-id* ;

using-declaration
template-declaration

member-declarator-list:
member-declarator
member-declarator-list , *member-declarator*

member-declarator:
declarator pure-specifier$_{opt}$
declarator constant-initializer$_{opt}$
identifier$_{opt}$: *constant-expression*

pure-specifier:
= 0

constant-initializer:
= *constant-expression*

Ради совместимости с языком C допускается в одной и той же области видимости объявлять класс и не-класс с одним и тем же именем (§5.7). Например:

```
struct stat {/*...*/};
int stat(char* name, struct stat* buf);
```

В этом случае, имя ***stat*** — это не имя класса; на класс нужно ссылаться с помощью классового префикса.

Константные выражения определяются в §C.5.

А.8.1. Производные классы

См. главу 12 и главу 15.

base-clause:
: *base-specifier-list*

base-specifier-list:
base-specifier
base-specifier-list , *base-specifier*

base-specifier:
::$_{opt}$ *nested-name-specifier*$_{opt}$ *class-name*
`virtual` *access-specifier*$_{opt}$ ::$_{opt}$ *nested-name-specifier*$_{opt}$ *class-name*
access-specifier `virtual`$_{opt}$ ::$_{opt}$ *nested-name-specifier*$_{opt}$ *class-name*

access-specifier:
`private`
`protected`
`public`

А.8.2. Особые функции-члены

См. §11.4 (операции преобразования), §10.4.6 (инициализация членов класса) и §12.2.2 (инициализация базового класса).

conversion-function-id:
`operator` *conversion-type-id*

conversion-type-id:
 type-specifier-seq conversion-declarator$_{opt}$

conversion-declarator:
 ptr-operator conversion-declarator$_{opt}$

ctor-initializer:
 : *mem-initializer-list*

mem-initializer-list:
 mem-initializer
 mem-initializer , *mem-initializer-list*

mem-initializer:
 mem-initializer-id (*expression-list*$_{opt}$)

mem-initializer-id:
 ::$_{opt}$ *nested-name-specifier*$_{opt}$ *class-name*
 identifier

А.8.3. Перегрузка

См. главу 11.

operator-function-id:
 operator *operator*

operator: one of

```
new     delete     new[]     delete[]
+       -     *    /    %    ^    &    |    ~    !    =    <    >
+=      -=    *=   /=   %=   ^=   &=   |=   <<   >>   >>=  <<=  ==
!=      <=    >=   &&   ||   ++   --   ,    ->*  ->   ()   []
```

А.9. Шаблоны

Шаблоны объясняются в главе 13 и §С.13.

template-declaration:
 export$_{opt}$ template < *template-parameter-list* > *declaration*

template-parameter-list:
 template-parameter
 template-parameter-list , *template-parameter*

template-parameter:
 type-parameter
 parameter-declaration

type-parameter:
 class *identifier*$_{opt}$
 class *identifier*$_{opt}$ = *type-id*
 typename *identifier*$_{opt}$
 typename *identifier*$_{opt}$ = *type-id*
 template < *template-parameter-list* > class *identifier*$_{opt}$
 template < *template-parameter-list* > class *identifier*$_{opt}$ = *id-expression*

template-id:
  template-name < *template-argument-list*$_{opt}$ >

template-name:
  identifier

template-argument-list:
  template-argument
  template-argument-list , *template-argument*

template-argument:
  assignment-expression
  type-id
  id-expression

explicit-instantiation:
  `template` *declaration*

explicit-specialization:
  `template < >` *declaration*

Явная специализация аргументов шаблона открывает возможность запутанной синтаксической неоднозначности. Рассмотрим пример:

```
void h()
{
 f<1>(0); // неоднозначность: ((f)<1)>(0) или (f<1>)(0)?
          // разрешение неоднозначности: вызывается f<1> с аргументом 0
}
```

Разрешение неоднозначности просто и эффективно: если ***f*** — имя шаблона, то *f*< — это начало квалифицированного имени шаблона, так что последующие лексемы должны интерпретироваться отталкиваясь от этого факта; в противном случае < означает знак «меньше, чем». Аналогично, первый невложенный знак > оканчивает список аргументов шаблона. Если требуется знак «больше, чем», то следует применить круглые скобки:

```
f< a>b >(0);       // синтаксическая ошибка
f< (a>b) >(0);     // ok
```

Подобная лексическая неоднозначность возникает, когда оканчивающие знаки > расположены вплотную друг к другу. Например:

```
list<vector<int>> lv1;     // error: неожиданная операция >> (сдвиг вправо)
list< vector<int> > lv2;   // правильно: список векторов
```

Обратите внимание на пробел между двумя знаками >; >> — это операция сдвига. Это может стать источником путаницы.

А.10. Обработка исключений

См. §8.3 и главу 14.

try-block:
  `try` *compound-statement handler-seq*

function-try-block:
`try` *ctor-initializer*$_{opt}$ *function-body handler-seq*

handler-seq:
handler handler-seq$_{opt}$

handler:
`catch (` *exception-declaration* `)` *compound-statement*

exception-declaration:
type-specifier-seq declarator
type-specifier-seq abstract-declarator
type-specifier-seq
`...`

throw-expression:
`throw` *assignment-expression*$_{opt}$

exception-specification:
`throw (` *type-id-list*$_{opt}$ `)`

type-id-list:
type-id
type-id-list `,` *type-id*

А.11. Директивы препроцессора

Препроцессор — это относительно простой макрообработчик, работающий в основном с лексемами, а не с отдельными символами. Дополнительно к возможности определять и использовать макросы (§7.8) препроцессор обеспечивает механизмы для включения текстовых файлов и стандартных заголовочных файлов (§9.2.1), а также для условной компиляции, базирующейся на макроопределениях (§9.3.3). Например:

```
#if OPT==4
#include "header4.h"
#elif 0<OPT
#include "someheader.h"
#else
#include <cstdlib>
#endif
```

Все директивы препроцессора начинаются с символа #, который должен быть первым не пробельным символом на своей строке.

preprocessing-file:
group$_{opt}$

group:
group-part
group group-part

group-part:
pp-tokens$_{opt}$ *new-line*
if-section
control-line

if-section:
  if-group elif-groups$_{opt}$ *else-group*$_{opt}$ *endif-line*

if group:
  # `if` *constant-expression new-line group*$_{opt}$
  # `ifdef` *identifier new-line group*$_{opt}$
  # `ifndef` *identifier new-line group*$_{opt}$

elif- groups:
  # elif-group
  elif-groups elif-group

elif-group:
  # `elif` *constant-expression new-line group*$_{opt}$

else-group:
  # `else` *new-line group*$_{opt}$

endif-line:
  # `endif` *new-line*

control-line:
  # `include` *pp-tokens new-line*
  # `define` *identifier replacement-list new-line*
  # `define` *identifier lparen identifier-list*$_{opt}$) *replacement-list new-line*
  # `undef` *identifier new-line*
  # `line` *pp-tokens new-line*
  # `error` *pp-tokens*$_{opt}$ *new-line*
  # `pragma` *pp-tokens*$_{opt}$ *new-line*
  # new-line

lparen:
  the left-parenthesis character without preceding white-space

replacement-list:
  pp-tokens$_{opt}$

pp-tokens:
  preprocessing-token
  pp-tokens preprocessing-token

new-line:
  the new-line character

identifier-list:
  identifier
  identifier-list , identifier

Приложение В

Совместимость

Вы следуйте своим обычаям, а я буду следовать своим.
— Ч. Нэпьер

Совместимость C и C++ — «тихие» отличия — код на C, не являющийся C++-кодом — нежелательные особенности — код на C++, не являющийся кодом на C —старые реализации C++ — заголовочные файлы — стандартная библиотека — пространства имен — ошибки выделения памяти — шаблоны — инициализаторы в операторах ***for*** — советы — упражнения.

В.1. Введение

В данном приложении обсуждаются расхождения между C и C++, а также между стандартом C++ (ISO/IEC 14882) и более ранними реализациями языка C++. Мы хотим документировать эти различия, которые могут создать для программиста некоторые проблемы, а также указать на пути их преодоления. Большинство таких проблем возникают, когда пытаются перевести C-программу на язык C++, когда осуществляют переход с одной из ранних версий C++ на другую или когда компилируют современную C++-программу старым компилятором. Моя цель — не утопить вас в огромной массе всех потенциально возможных проблем совместимости с каждой из существующих версий C++, а, скорее, перечислить наиболее типичные проблемы и представить их стандартные решения.

При рассмотрении вопросов совместимости очень важно учесть диапазон реализаций, при которых программа должна работать. Для изучения C++ целесообразно выбрать наиболее полную и удобную реализацию. Для производства конечного отчуждаемого продукта больше подходит консервативная стратегия, призванная максимизировать число систем, с которыми продукт будет работать. В прошлом это служило оправданием бегства от новшеств C++. Однако теперь реализации сближаются и необходимость в переносимости с одной из них на другую уже не требует такой осторожности, как раньше.

B.2. Совместимость C и C++

За минимальными исключениями язык C (имеется в виду стандарт C89, ISO/IEC 9899:1990) является подмножеством C++. Наиболее значимые различия возникают из-за большей приверженности C++ статической проверке типов. Хорошо написанные на C программы часто оказываются и допустимыми программами на C++. Компилятор выявляет все различия между C и C++.

B.2.1. «Тихие» отличия

За небольшими исключениями программы на C и C++ имеют одинаковый смысл. По счастью, эти исключения («тихие» отличия) непринципиальны.

В языке C размер символьной константы и перечислений равен sizeof(int). В C++ *sizeof*('*a*') равен *sizeof*(*char*), а размер перечислений определяется конкретной реализацией (§4.8).

В языке C++ есть комментарии //, а в языке C их официально нет (но многие реализации их допускают). Это отличие можно использовать для написания программы, ведущей себя на разных языках по-разному. Например:

```
int f(int a, int b)
{
  return a //*  маловероятно */ b
  ;          /* нереалистично: точка с запятой на отдельной строке во избежание
              синтаксической ошибки */
}
```

Международный стандарт языка C пересматривается с целью введения комментариев //.

Имя структуры, объявленной во вложенной области видимости, может скрыть имя объекта, функции, перечисления или типа из объемлющей области видимости. Например:

```
int x[99];
void f()
{
  struct x {int a;};
  sizeof(x); /* в C - размер массива; в C++ - размер структуры */
}
```

B.2.2. Код на C, не являющийся C++-кодом

Разночтения C/C++, которые вызывают большинство проблем, не столь уж и сложны. Большинство их них легко обнаруживаются компилятором. В данном разделе приведены примеры на C, которые не соответствуют языку C++. Многие из них считаются устаревшими для современного C.

В языке C большинство функций можно вызывать без предварительного объявления:

```
main()                               /* не C++. В C - плохой стиль */
{
  double sq2 = sqrt(2);              /* вызов необъявленной функции */
```

```
  printf("the square root of 2 is %g\n", sq2);    /* вызов необъявленной функции */
}
```

Последовательное использование предварительных объявлений (прототипов функций) рекомендуется и для языка С. Когда этому следуют на практике (или когда компилятор имеет установки следить за этим), то проблем с переходом от С к С++ не возникает. Если же вызываются необъявленные функции, то нужно очень точно знать правила языка С, чтобы понять, где возникла проблема с переносимостью. Например, функция *main*() из предыдущего примера содержит с точки зрения языка С по крайней мере две ошибки.

В языке С функция, объявленная без аргументов, может принимать любое количество аргументов любого типа. Такой стиль стандартом С признан устаревшим, но он не так уж и редко встречается в реальных программах:

```
void f();        /* не упомянуты типы аргументов */

void g()
{
 f(2);           /* Не С++. Плохой стиль в С */
}
```

В языке С функции можно определять в стиле, когда типы объявляются после перечисления аргументов:

```
void f(a,p,c) char *p; char c; { /* ... */ }  /* С. Не С++ */
```

Это лучше переписать в виде

```
void f(int a, char* p, char c) { /* ... */ }
```

В языке С и в старых версиях С++ (предшествовавших принятию стандарта) типом по умолчанию является *int*. Например:

```
const a = 7;    /* В С подразумевается int. Не С++ */
```

Поздняя спецификация С99 запрещает "умолчательные *int*", как это имеет место в стандарте языке С++.

Язык С допускает определения структур в объявлениях возврата функций или их аргументов. Например:

```
struct S {int x,y; } f();           /*  С. Не С++ */
void g(struct S {int x,y; } y);     /*  С. Не С++ */
```

С точки зрения правил языка С++ для определения типов такие объявления бесполезны, а потому недопустимы.

В С переменным перечислительного типа можно присваивать целые значения:

```
enum Direction {up,down};
enum Direction d = 1;          /* error: int присваивается Direction; ok в С */
```

В С++ гораздо больше ключевых слов, чем в С. Если какое-либо из них присутствует в С-программе как идентификатор, то во имя совместимости это нужно устранить.

Ключевые слова в C++ (но не в C)					
and	*and_eq*	*asm*	*bitand*	*bitor*	*bool*
catch	*class*	*compl*	*const_cast*	*delete*	*dynamic_cast*
explicit	*export*	*false*	*friend*	*inline*	*mutable*
namespace	*new*	*not*	*not_eq*	*operator*	*or*
or_eq	*private*	*protected*	*public*	*reinterpret_cast*	*static_cast*
template	*this*	*throw*	*true*	*try*	*typeid*
typename	*using*	*virtual*	*wchar_t*	*xor*	*xor_eq*

В C некоторые из ключевых слов C++ являются макросами, определенными в стандартных заголовочных файлах:

Ключевые слова C++ (макросы в C)							
and	*and_eq*	*bitand*	*bitor*	*bool*	*compl*	*false*	
not	*not_eq*	*or*	*or_eq*	*true*	*wchar_t*	*xor*	*xor_eq*

То есть в C они могут участвовать в проверках условных препроцессорных директив *#ifdef*, переопределяться и т.д.

В языке C глобальные объекты данных могут несколько раз объявляться в одной и той же единице трансляции без использования спецификатора ***extern***. Если инициализатор присутствует только в одном из них, то объект считается определенным один раз. Например:

```
int i; int i; /* определяет или объявляет единственную целую переменную i; не C++ */
```

В C++ любая сущность определяется один раз (§9.2.3).

В языке C++ класс не может иметь имя, совпадающее с именем, определенным оператором ***typedef*** для ссылки на другой тип в той же самой области видимости (§5.7).

В языке C тип ***void**** можно использовать в правой части операции присваивания или при инициализации переменной любого указательного типа; в C++ этого делать нельзя (§5.6). Например:

```
void f(int n)
{
  int* p = malloc(n*sizeof(int)); /* не C++ */
}
```

Язык C разрешает переходы ***goto*** к помеченным метками операторам (§А.6) в обход инициализации, что запрещает делать язык C++.

В языке C глобальная константа по определению имеет внешнее связывание; в C++ это не так и ее требуется инициализировать, если только она не объявляется явно с модификатором ***extern*** (§5.4).

В языке C имена вложенных структур располагаются в той же области видимости, что и объемлющая структура. Например:

```
struct S
{
  struct T { /* . . . */ } ;
  // ...
} ;

struct T x;                  /* ok в С - означает S::T x; Не С++ */
```

В С массив может инициализироваться инициализатором, в котором значений больше, чем элементов у массива. Например:

```
char v[5] = "Oscar" ;   /* ok в С, терминальный нуль не используется. Не С++ */
```

В.2.3. Нежелательные особенности

Помечая ту или иную особенность языка как *нежелательную* (*deprecated*), комитет по стандартизации выражает желание устранить ее. Однако у комитета нет права удалить из языка широко используемые средства (как бы избыточны или опасны они не были). Эти пометки служат настоятельным советом программисту избегать помеченные особенности.

Ключевое слово **static**, которое в общем случае означает «статическое выделение памяти», может использоваться и для индикации факта, что функция или объект локальны по отношению к единице трансляции. Например:

```
// file1:
static int glob;

// file2:
static int glob;
```

Эта программа имеет две целые переменные с именем **glob**. Каждая из этих **glob** используется исключительно функциями из их единицы трансляции.

Применение **static** для отражения свойства «локально для единицы трансляции» помечено в С++ как нежелательное (deprecated). Вместо этого следует применять неименованные пространства имен (§8.2.5.1).

Неявное приведение из строкового литерала в (неконстантный) **char*** также объявлено нежелательным. Используйте именованные массивы элементов типа **char**, или избегайте присваивания строковых литералов переменным типа **char*** (§5.2.2).

Приведение типа в стиле языка С также помечено как нежелательное, ибо в языке С++ введены новые операции приведения типа. Программисты должны отнестись к этому серьезно. Там, где требуются явные приведения типа, операции **static_cast**, **reinterpret_cast**, **const_cast** или их комбинации делают все необходимое. Новые операции приведения (по сравнению с приведением в стиле С) точнее отражают их назначение и более заметны в тексте программы (§6.2.7).

В.2.4. Код на С++, не являющийся кодом на С

Здесь перечисляются средства С++, отсутствующие в языке С. Они отсортированы по назначению. На самом деле, их можно классифицировать по-разному и многие средства служат нескольким целям, так что к данной классификации не стоит относиться слишком серьезно.

- Средства, предназначенные в первую очередь для удобства записи:
  1. Комментарии // (§2.3); добавляются в C99
  2. Поддержка ограниченных наборов символов (§C.3.1); частично добавляется в C99
  3. Поддержка расширенных наборов символов (§C.3.3); частично добавляется в C99
  4. Неконстантные инициализаторы для объектов со статическим хранением (§9.4.1)
  5. ***const*** в константных выражениях (§5.4, §C.5)
  6. Объявления как операторы (§6.3.1); добавляются в C99
  7. Объявления в инициализаторах операторов ***for*** (§6.3.3); добавляются в C99
  8. Объявления в условиях (§6.3.2.1)
  9. Имена структур не нужно предварять словом ***struct*** (§5.7)
- Средства, предназначенные в первую очередь для усиления системы типов:
  1. Проверка типов аргументов функций (§7.1); добавлено в C позже (§B.2.2)
  2. Безопасная по типам компоновка (§9.2, §9.2.3)
  3. Выделение свободной памяти операциями ***new*** и ***delete*** (§6.2.6, §10.4.5, §15.6)
  4. ***const*** (§5.4, §5.4.1); добавлено в C позже
  5. Логический тип ***bool*** (§4.2); частично добавляется в C99
  6. Новый синтаксис приведения типов (§6.2.7)
- Средства, связанные с пользовательскими типами:
  1. Классы (глава 10)
  2. Функции-члены (§10.2.1) и вложенные классы (§11.12)
  3. Конструкторы и деструкторы (§10.2.3, §10.4.1)
  4. Производные классы (глава 12, глава 15)
  5. Виртуальные функции и абстрактные классы (§12.2.6, §12.3)
  6. Режимы доступа ***public/protected/private*** (§10.2.2, §15.3, §C.11)
  7. Дружественные функции (§11.5)
  8. Указатели на члены классов (§15.5, §C.12)
  9. Статические члены (§10.2.4)
  10. Ключевое слово ***mutable*** (§10.2.7.2)
  11. Перегрузка операций (глава 11)
  12. Ссылки (§5.5)
- Средства, предназначенные в первую очередь для организации программ:
  1. Шаблоны (глава 13, §C.13)
  2. Встраиваемые функции (§7.1.1); добавляются в C99
  3. Аргументы по умолчанию (§7.5)
  4. Перегрузка функций (§7.4)

5. Пространства имен (§8.2)
6. Операция разрешения области видимости (операция ::, §4.9.4)
7. Обработка исключений (§8.3; глава 14)
8. Средства RTTI (§15.4)

Ключевые слова, добавленные в C++ (§B.2.2), можно использовать для идентификации большинства средств, специфичных для этого языка. В то же время, некоторые средства C++, такие как перегрузка функций или константы в ***константных*** выражениях, не идентифицируются по ключевым словам. Кроме перечисленных особенностей стандартная библиотека C++ (§16.1.2) по большей части специфична именно для C++.

Макрос ***__cplusplus*** применяется для того, чтобы выбрать компилятор C или C++ (§9.2.4).

В.3. Старые реализации C++

Язык C++ интенсивно используется с 1983 года (§1.4). С тех пор сменилось несколько его версий и множество независимо выполненных реализаций. Фундаментальная цель стандартизации языка заключалась в том, чтобы дать разработчикам реализаций и пользователям единое определение C++. В то же время, пока стандартное определение не распространится широко в среде программистов, придется смириться с неприятным фактом, что не каждая реализация поддерживает все средства, описанные в этой книге.

К сожалению, часто люди, начинающие изучать язык C++, сталкиваются с реализациями пятилетней давности. Как правило, это объясняется их широкой распространенностью и бесплатностью. Имея выбор, ни один профессионал не станет работать на таком антиквариате. Для новичков же применение устаревших компиляторов оборачивается серьезными скрытыми потерями. Например, недостаток современных языковых средств и библиотек заставляет их бороться с проблемами, которые уже были успешно решены в новых реализациях. Кроме того, наносится серьезный вред стилю написания программ и порождает неточный взгляд на то, каков же язык C++ на самом деле. Наилучшее подмножество для первичного изучения и использования вовсе не сводится к набору низкоуровневых средств (и не к общему подмножеству языков C и C++; §1.2). Чтобы облегчить изучение и получить правильное представление о языке C++, я рекомендую опереться на стандартную библиотеку и шаблоны.

Первая коммерческая версия C++ появилась в 1985 году. Она соответствовала описанию языка, представленному в первом издании настоящей книги — там не было множественного наследования, шаблонов, средств RTTI, исключений и пространств имен. Сегодня я не вижу смысла пользоваться реализацией, которая не поддерживала бы хотя бы некоторой части из перечисленных свойств. В 1989 году я добавил к языку множественное наследование, шаблоны и исключения. К сожалению, ранняя поддержка шаблонов и исключений оставляла желать лучшего. Если ваша реализация относится к этой фазе развития C++ — срочно выполните ее обновление до современного состояния.

В общем случае разумно применять компиляторы и среды, более-менее соответствующие стандарту и минимально опирающиеся на неопределенные, зависящие

от реализации средства. Выполняйте проектирование в предположении, что вам доступны все современные возможности, и только после этого разыскивайте необходимые компиляторы и инструменты. Это помогает улучшить структуру программы и облегчить ее сопровождение по сравнению со случаем, когда применяется минимальное общее подмножество языка. Также будьте осторожны с применением зависящих от реализации расширений языка и делайте это лишь в случае крайней необходимости.

В.3.1. Заголовочные файлы

Традиционно все заголовочные файлы имели расширение *.h*. То есть реализации C++ предоставляли заголовочные файлы *<map.h>*, *<iostream.h>* и т.д. А для совместимости они так делают и сейчас.

Когда комитету по стандартизации понадобились заголовочные файлы для пересмотренной версии стандартной библиотеки и для новых библиотечных средств, их именование превратилось в небольшую проблему — использование старых *h*-имен вступило бы в конфликт с задачей поддержания совместимости. Кроме того, применение суффиксов *h* все равно избыточно, так как <> и без того указывают на стандартные заголовочные файлы.

В итоге, стандартная библиотека предоставляет заголовочные файлы с бессуффиксными именами, такими как *<map>* и *<iostream>*. Объявления в этих файлах помещены в пространство имен *std*, в то время как старые заголовочные файлы (с расширениями *h*) размещают объявления в глобальном пространстве имен. Рассмотрим пример:

```
#include <iostream>

int main ()
{
  std::cout<< "Hello, world!\n";
}
```

Если у вас не получается скомпилировать этот пример, попробуйте более традиционную версию:

```
#include <iostream.h>

int main ()
{
  cout << "Hello, world!\n";
}
```

Некоторые из наиболее серьезных проблем с переносимостью возникают из-за непереносимых заголовочных файлов, причем стандартные заголовочные файлы вносят здесь лишь малую долю. Часто программы зависят от множества заголовочных файлов, присутствующих не на каждой системе, от объявлений, которые на разных системах располагаются в разных заголовочных файлах, и от объявлений, кажущихся стандартными (они располагаются в заголовочных файлах со стандартными именами), но не входящими ни в какие стандарты.

Полностью удовлетворительных решений проблемы несовместимости заголовочных файлов не существует. Самая общая идея заключается в максимальной локализации (компактизации) различий и введения дополнительного уров-

ня косвенности, дабы избежать прямых зависимостей от несовместимых заголовочных файлов. К примеру, пусть необходимые нам объявления на разных системах располагаются в разных заголовочных файлах. Тогда можно включить в приложение единственный промежуточный (специфичный для данного приложения) заголовочный файл, который, в свою очередь, будет включать разные заголовочные файлы для разных систем. Аналогично, если некоторая функциональность немного по-разному предоставляется на разных системах, мы можем обращаться к ней через специфичные для приложения интерфейсные классы и функции.

B.3.2. Стандартная библиотека

Неудивительно, что реализации библиотек C++, предшествовавших стандарту, могли не содержать некоторых частей стандартной библиотеки. В большинстве своем они будут содержать потоки ввода/вывода, нешаблонный класс ***complex***, те или иные строковые классы, а также библиотеку языка С. Как правило, будут отсутствовать классы ***map***, ***list***, ***valarray*** и т.п. Применяйте в таком случае сторонние библиотеки, допускающие переход к стандарту при обновлении вашей системы до современного состояния. Но в любом случае лучше использовать нестандартные классы строк, списков и ассоциативных массивов, чем возвращаться к низкоуровневому программированию в стиле С. Имейте также в виду, что доступны для бесплатного использования хорошие реализации STL-части стандартной библиотеки (главы 16, 17, 18 и 19).

Ранние реализации стандартной библиотеки также не совсем полны. Например, некоторые контейнеры могут не поддерживать аллокаторов (распределителей памяти), а другие могут требовать явного указания аллокаторов для каждого класса. Похожие проблемы могут касаться аргументов, определяющих политику алгоритма, например критерий сравнения:

```
list<int> li;                     // ok, но некоторые реализации требуют аллокатора
list<int, allocator<int> > li2;   // ok, но некоторые реализации не реализуют аллокаторы

map<string, Record> m1;           // ok, но некоторые реализации требуют операцию <
map<string, Record, less<string> > m2;
```

Пользуйтесь той версией, которую допускает ваша реализация.

Ранние реализации C++ предоставляли ***istrstream*** и ***ostrstream*** в ***<strstream.h>***, а не ***istringstream*** и ***ostringstream*** из ***<sstream>***, причем работали эти потоки напрямую с ***char***[] (см. §21.10[26]).

Достандартные потоки не были параметризуемыми. В частности, шаблоны с префиксом ***basic_***, появились только в стандарте, класс ***basic_ios*** назывался ***ios***, а ***iostate*** назывался ***io_state***.

B.3.3. Пространства имен

Если ваша реализация не поддерживает пространств имен, отражайте логическую структуру программы соответствующей структурой исходных файлов (глава 9). Аналогично, применяйте заголовочные файлы для отражения интерфейсов, предоставляемых разным реализациям, а также для языка С.

В отсутствие пространств имен вообще используйте *static* для компенсации отсутствия неименованного пространства имен. Также для глобальных имен применяйте идентифицирующие их префиксы:

```
class bs_string {/*...*/};
typedef int bs_bool;
class joe_string;
enum joe_bool {joe_false,joe_true};
```

Внимательно выбирайте префикс. Существующие библиотеки С и С++ переполнены разными префиксами.

В.3.4. Ошибки выделения памяти

До появления в языке С++ обработки исключений операция new возвращала нуль, когда не удавалось выделить память. В стандарте С++ операция new по умолчанию генерирует в таком случае исключение *bad_alloc*. В любом случае, перехватывая *bad_alloc* или реагируя на возврат нуля, трудно для большинства систем предложить что-то более интересное, чем просто сообщение об ошибке.

Когда по каким-то причинам нужно придерживаться старого способа индикации об ошибке выделения памяти, то в случае, если не задан *_new_handler*, применение аллокатора *nothrow* заставляет возвращать нуль в ошибочных случаях:

```
X* p1 = new X;              // генерируется bad_alloc при отсутствии необходимой памяти
X* p2 = new (nothrow) X;    // возвращает 0 при отсутствии необходимой памяти
```

В.3.5. Шаблоны

Стандарт С++ ввел некоторые новые элементы шаблонов и уточнил работу со старыми (ранее имевшимися) средствами.

Если ваша реализация не поддерживает частичную специализацию, используйте для шаблона иное имя (в противном случае он был бы специализацией):

```
template<class T> class plist: private list<void*>
{
  //...
};
```

Если ваша реализация не поддерживает шаблонных членов шаблонов, некоторые приемы программирования станут недостижимыми. В частности, шаблонные члены шаблонов позволяют программистам выполнять конструирование и преобразование типа с гибкостью, недостижимой иными способами (§13.6.2). Рассмотрим пример:

```
template<class T> class X
{
  //...
  template<class A> X(const A& a);
};
```

В отсутствие шаблонных членов шаблонов этот пример придется переделать так, чтобы ограничить себя набором определенных типов:

```
template<class T> class X
{
  // ...
  X(const A1& a);
  X(const A2& a);
  // ...
};
```

В большинстве ранних реализаций для всех функций-членов шаблонов генерировался код в момент конкретизации шаблона (template instantiation). Это могло приводить к ошибкам для неиспользуемых функций-членов (§C.13.9.1). Решение состоит в том, чтобы поместить определение функций-членов позже объявления класса. Например, вместо

```
template<class T> class Container
{
  // ...
public:
  void sort() { /* используем < */ }
};

class Glob { /* для Glob нет операции < */ };
Container<Glob> cg;   // некоторые старые реализации пытаются определить
                      // Container<Glob>::sort();
```

используйте

```
template<class T> class Container
{
  // ...
public:
  void sort();
};

template<class T> void Container<T>::sort() { /* используем < */ }

class Glob { /* для Glob нет операции < */ };
Container<Glob> cg;   // нет проблем до тех пор, пока не вызывается cg.sort()
```

Кроме того, ранние реализации не допускали использования членов классовых шаблонов, объявленных позже точки использования, как в следующем примере:

```
template<class T> class Vector
{
public:
  T& operator[](size_t i) { return v[i]; }     // v объявляется ниже
  // ...

private:
  T* v;                                        // не найдено!
  size_t sz;
};
```

Тогда нужно либо переупорядочить объявления членов шаблона, либо расположить определения функций-членов после объявления шаблона класса.

Некоторые из ранних реализаций не понимают умолчательных значений для параметров шаблона (§13.4:1). В таком случае каждому формальному параметру типа шаблона нужно в обязательном порядке сопоставить явный аргумент. Например:

```
template<class Key, class T, class LT = less<T> > class map
{
  //...
};

map<string,int> m;                   // Oops: умолчательные аргументы не реализованы
map<string,int,less<string> > m2;   // обход проблемы: все указываем явно
```

В.3.6. Инициализаторы в операторах for

Рассмотрим следующий код:

```
void f(vector<char>& v, int m)
{
  for(int i=0; i<v.size() && i<=m; ++i) cout << v[i];
  if(i == m)     // error: i недостижимо после конца for
  {
    //...
  }
}
```

Такой код обычно работал, поскольку в изначальном определении C++ область видимости переменной цикла совпадала с областью видимости, в которую входил сам оператор *for*. Если вам встретится такой код, просто расположите определение переменной цикла перед оператором *for*:

```
void f2 (vector<char>& v, int m)
{
  int i= 0;

  for(; i<v.size() && i<=m; ++i) cout << v[i];
  if(i==m)
  {
    //...
  }
}
```

В.4. Советы

1. Для изучения C++ применяйте самые современные и самые полные из доступных вам реализаций стандарта C++; §В.3.
2. Общее подмножество языков C и C++ — не лучшее подмножество для первоначального изучения C++; §1.6, §В.3.
3. Помните, что не все реализации C++ поддерживают его современный стандарт. Перед тем, как использовать в производственном коде какую-либо из новейших черт языка, опробуйте ее сначала на тестовых примерах, чтобы убедиться в поддержке стандарта вашей реализацией и уровнем достигаемой при этом производительности; §8.5[6-7], §16.5[10], §В.5[7].

4. Избегайте особенностей языка, помеченных как нежелательные (deprecated), например статических глобальных переменных. Избегайте приведения типа в С-стиле; §6.2.7, §В.2.3.
5. «Умолчательные ***int***» запрещены — явно определяйте типы для функций, переменных, констант и т.д.; §В.2.2.
6. Преобразуя программу из языка С в язык С++, последовательно используйте прототипы функций и стандартные заголовочные файлы; §В.2.2.
7. Преобразуя программу из языка С в язык С++, переименуйте переменные, совпадающие с ключевыми словами языка С++; §В.2.2.
8. Преобразуя программу из языка С в язык С++, возвраты ***malloc***() приводите к нужному типу, или замените вызовы ***malloc***() на операцию new; §В.2.2.
9. При переходе от ***malloc***() и ***free***() к ***new*** и ***delete***, подумайте о применении ***vector***, ***push_back***() и ***reserve***() вместо ***realloc***(); §3.8, §16.3.5.
10. Преобразуя программу из языка С в язык С++, помните о том, что нет неявного преобразования из ***int*** в перечисления; при необходимости используйте явные преобразования; §4.8.
11. Объекты из пространства имен ***std*** определяются в бессуффиксных заголовочных файлах (например, ***std::cout*** определяется в файле <***iostream***>). Старые реализации объявляют средства стандартной библиотеки в глобальном пространстве имен и располагают их в заголовочных файлах с суффиксом ***.h*** (***::cout*** определяется в <***iostream.h***>); §9.2.2, §В.3.1.
12. Если старый код проверяет возврат операции new на нуль, то нужно его исправить в сторону перехвата исключения ***bad_alloc***, или применить ***new***(***nothrow***); §В.3.4.
13. Если ваша реализация не поддерживает умолчательных значений параметров шаблона, указывайте аргументы явно. Часто с помощью оператора ***typedef*** можно избежать повторных указаний аргументов (аналогично тому, как ***typedef string*** помогает избежать написания ***basic_string***<***char***, ***char_traits*** <***char***>, ***allocator***<***char***> >); §В.3.5.
14. Используйте <***string***> для доступа к ***std::string*** (<***string.h***> содержит функции для работы с С-строками); §9.2.2, §В.3.1.
15. Для каждого стандартного заголовочного файла <***X.h***>, помещающего объявления в глобальное пространство имен, заголовочный файл <***cX***> помещает эти имена в пространство имен ***std***; §В.3.1.
16. Многие системы имеют заголовочный файл "***String.h***", определяющий строковый тип. Отметим, что такие строки отличаются от строк ***string*** стандартной библиотеки.
17. Предпочитайте стандартные средства нестандартным; §20.1, §В.3, §С.2.
18. Используйте ***extern*** "***C***" при объявлении С-функций; §9.2.4.

В.5. Упражнения

1. (*2.5) Возьмите С-программу и преобразуйте ее в программу на С++. Составьте список используемых конструкций, не являющихся конструкциями языка С++, и определите, соответствуют ли они стандарту ANSI С. Сначала приведите программу в строгое соответствие с ANSI С (прототипы и т.д.), а затем — с С++. Оцените время, необходимое для подобных преобразований в случае программы, содержащей 100000 строк кода.
2. (*2.5) Напишите программу, помогающую конвертировать С-программы на язык С++ путем переименовывания переменных, совпадающих с ключевыми словами С++, путем замены ***malloc***() на операции ***new*** и т.д. Намек: не пытайтесь достичь полного совершенства.
3. (*2) Замените все вызовы ***malloc***() в С++-программах, написанных в стиле языка С, на операции ***new***. Подсказка: §В.4[8-9].
4. (*2.5) В С++-программе, написанной в стиле языка С, минимизируйте использование макросов, глобальных переменных, неинициализированных переменных и приведений типа в С-стиле.
5. (*3) Возьмите С++-программу, полученную в результате грубой переделки из С-программы, и покритикуйте ее на предмет локализации информации, абстракции, читабельности, расширяемости и возможности повторного использования ее частей. Выполните хотя бы одно существенное изменение программы, отталкиваясь от вашей критики.
6. (*2) Возьмите маленькую (скажем, строк на 500) С++-программу и преобразуйте ее в программу на С. Сравните оригинал и полученный результат по размеру и возможностям сопровождения.
7. (*3) Напишите небольшой набор тестовых программ для выяснения, поддерживает ли ваша С++-реализация самые последние стандарты. Например, какова область видимости переменной, определенной в инициализирующей секции оператора for (§В.3.6)?, поддерживаются ли умолчательные значения параметров шаблонов (§В.3.5)?, поддерживаются ли шаблонные члены шаблонов (§13.6.2)?, поддерживается ли поиск имен по аргументам (§8.2.6)? Подсказка: §В.2.4.
8. (*2.5) Возьмите С++-программу, использующую заголовочный файл <***X.h***> и преобразуйте ее так, чтобы она использовала заголовочные файлы <***X***> и <***cX***>. Минимизируйте применение директив ***using***.

Приложение С

Технические подробности

Глубоко в сердце сознания и Вселенной — есть смысл.
— Слартибартфаст

Стандарт — символьные наборы — целые литералы — константные выражения — продвижения и преобразования — многомерные массивы — битовые поля и объединения — управление памятью — автоматическая сборка мусора — пространства имен — контроль доступа — указатели на члены данных — шаблоны — *статические* члены — друзья — шаблоны в качестве параметров шаблона — выведение аргумента шаблона — шаблоны и ключевое слово ***typename*** — конкретизация — связывание имен — шаблоны и пространства имен — явная конкретизация — советы.

C.1. Введение и обзор

Данная глава содержит технические детали и примеры, которые не совсем вписываются в то, как я представляю основные свойства языка C++ и их применение. Эти технические детали важны при реальном написании программ и еще более важны при чтении текстов ранее написанных программ, которые могут эти детали содержать. В то же время, я считаю эти детали техническими, поскольку они не должны отвлекать внимание студентов от важнейшей задачи первоначального изучения языка C++, а программистов — от написания программ в стиле, обеспечивающем (насколько это вообще возможно) ясное и непосредственное отражение проектных идей.

C.2. Стандарт

Вопреки устоявшемуся мнению, строгое следование стандарту языка и библиотеки не гарантирует хорошего исходного кода программы и ее переносимости. Стандарт ничего не говорит о том, плох или хорош конкретный фрагмент программы; он просто формулирует, на что программист может рассчитывать в разных реализациях, а на что не может. Вполне можно написать ужасную программу в полном

соответствии со стандартом, в то время как множество успешных программ могут применять особенности языка, не описываемые стандартом.

Много важных вещей стандарт отдает на откуп реализациям. В таких случаях реализации должны обеспечить хотя и специфическое, но четко определенное и хорошо документированное поведение языковых конструкций. Например:

```
unsigned char c1 = 64;    // однозначно: char всегда имеет по крайней мере 8 бит
                          // и всегда может содержать значение 64
unsigned char c2 = 1256;  // зависит от реализации: урезание в случае 8-битного char
```

Инициализация ***c1*** четко определена, поскольку тип ***char*** обязан содержать по крайней мере 8 бит. В то же время, инициализация ***c2*** зависит от реализации, так как точное количество бит в ***char*** определяется не стандартом, а реализацией. Например, если в ***char*** всего 8 бит, то значение ***1256*** усекается до ***232*** (§C.6.2.1). Большая часть зависящих от реализации свойств языка коррелирует с различиями в железе, на котором программе предстоит работать.

При написании реальных программ приходится считаться с их поведением, зависящим от реализации. Такова цена за возможность эффективно работать на разных системах. Например, язык был бы проще, если бы тип ***char*** был жестко привязан к 8 битам, а тип ***int*** — к 32 битам. Однако имеются системы с 16- или 32-битовыми символьными наборами, а целые числа в реальных задачах не умещаются в 32 бита — для дисков емкостью более *32 Гбайт* более уместны 48-битовые или 64-битовые числа.

Для улучшения переносимости лучше всего четко и явно обозначить места программы, зависящие от реализации, а особо тонкие различия тщательно локализовать и задокументировать. Обычная практика заключается в том, чтобы все зависимости размеров от аппаратуры представить в виде констант и определений типов в некотором заголовочном файле. Стандартная библиотека с этой целью предоставляет ***numeric_limits*** (§22.2).

Неопределенное поведение более коварно. Конструкция помечается стандартом как *неопределенная* (*undefined*), если невозможно потребовать от реализаций какого-либо разумного ее поведения. Как правило, программы, использующие конструкции с неопределенным поведением, ведут себя очень плохо. Например:

```
const int size = 4*1024;
char page[size];

void f()
{
  page[size+size] =7;         // не определено
}
```

Возможные последствия этого фрагмента могут включать в себя перезаписывание некоторых не имеющих отношения к нему данных и генерирование аппаратных исключений (ошибок). От реализации не требуется в таких случаях осуществлять выбор между возможными последствиями. Там, где используются мощные технологии оптимизации кода, последствия применения в программе конструкций с неопределенным поведением вообще не поддаются предсказанию. Если имеется набор очевидных и легко реализуемых альтернатив, конструкция считается зависящей от реализации, а не как имеющая неопределенное поведение.

В процессе разработки и написания программ стоит уделить серьезное внимание, силы и время, чтобы гарантировать отсутствие в программе конструкций, не

соответствующих стандарту. Часто это можно выполнить с помощью специализированных инструментальных средств.

С.3. Символьные наборы

Примеры в этой книге составлены с учетом американского варианта международного 7-битного символьного набора ISO 646-1983, названного ASCII (ANSI3.4-1968). Это может породить следующие проблемы у людей, работающих с C++ в средах с другими символьными наборами:

1. ASCII содержит символы пунктуации и символы операций (`[`, `{` и `!`), которые могут быть недопустимыми в других символьных наборах.
2. Требуются обозначения для символов, не имеющих удобного представления (например, «новая строка» или «символ с числовым кодом 17").
3. ASCII не содержит символов (ζ, λ и др.), применяемых в языках, отличных от английского.

С.3.1. Ограниченные наборы символов

ASCII символы `[`, `]`, `{`, `}`, `|` и `\` занимают в символьном наборе позиции, трактуемые ISO как алфавитные. В большинстве европейских символьных наборов ISO-646 эти позиции заняты буквами, которых в английском алфавите нет. Например, в датском языке тут присутствуют гласные ***Æ***, ***æ***, ***Ø***, ***ø***, ***Å è å***. Без этих символов по-датски не напишешь сколько-нибудь осмысленного текста.

Чтобы выражать национальные символы переносимым способом при помощи минимальных символьных наборов, вводятся *триграфы* (*trigraphs*). Это помогает распространению программ, но не облегчает чтение программ людьми. Долговременным решением проблемы могли бы стать локализованные (под национальные языки) средства разработки программ на C++. Но текущую помощь программистам в этом вопросе при использовании ограниченных символьных наборов предоставляют ключевые слова, *диграфы* (*digraphs*) и триграфы:

Ключевые слова		Диграфы		Триграфы	
and	`&&`	`<%`	`{`	`??=`	`#`
and_eq	`&=`	`%>`	`}`	`??(`	`[`
bitand	`&`	`<:`	`[`	`??<`	`{`
bitor	`|`	`:>`	`]`	`??/`	`\`
compl	`~`	`%:`	`#`	`??)`	`]`
not	`!`	`%:%:`	`##`	`??>`	`}`
or	`||`			`??*`	`^`
or_eq	`|=`			`??!`	`|`
xor	`^`			`??-`	`~`
xor_eq	`^=`				
not_eq	`!=`				

Программы с ключевыми словами и диграфами читаются лучше программ, использующих триграфы. Однако если символы, такие как {, недостижимы, то применение триграфов обязательно, дабы не потерять символы в строках и символьных константах. Например, '{' превращается в '??<'.

Некоторые люди вместо традиционных значков операций предпочитают использовать ключевые слова, например ***and***.

С.3.2. Escape-символы

Небольшое количество символов имеют имена, включающие обратную косую черту в качестве escape-символа:

Название	ASCII	C++
newline (перевод строки)	NL (LF)	`\n`
horizontal tab (горизон-я табуляция)	HT	`\t`
vertical tab (вертик-я табуляция)	VT	`\v`
backspace (забой)	BS	`\b`
carriage return (возврат каретки)	CR	`\r`
form feed (перевод страницы)	FF	`\f`
alert (звонок)	BEL	`\a`
backslash (обратная косая черта)	\	`\\`
question mark (знак вопроса)	?	`\?`
single quote (апостроф)	'	`\'`
double quote (двойная кавычка)	"	`\"`
octal number (восьмерич. число)	*ooo*	*ooo*
hex number (шестнадцатер. число)	*hhh*	*xhhh* ...

Несмотря на свой внешний вид, это одиночные символы.

Символ можно представить в виде одной, двух или трех восьмеричных цифр (с предшествующим \) или шестнадцатеричным числом (с предшествующим ***x***). На число шестнадцатеричных цифр в такой последовательности ограничений нет. Последовательность восьмеричных или шестнадцатеричных цифр обрывается на первом отличном от них символе. Например:

Восьмеричная	Шестнадцатеричная	Десятичная	ASCII
`'\6'`	`'\x6'`	6	ACK
`'\60'`	`'\x30'`	48	'0'
`'\137'`	`'\x05f'`	95	'_'

Это позволяет представить любой символ из машинного символьного набора и, в частности, вставить такие символы в символьные строки (см. §5.2.2). Применение

любых цифровых нотаций для символов делает программу непереносимой между машинами с разными символьными наборами.

Допускается записывать символьный литерал с применением нескольких символов, например '**ab**', но это считается архаичным, зависящим от реализации, так что этого лучше избегать.

Когда числовая константа в восьмеричной нотации внедряется в строку, целесообразно использовать для ее записи именно три цифры, иначе чтение такой строки вызывает затруднение (требуется следить за тем, цифра или нет стоит после числовой константы). Для шестнадцатеричных констант используйте две цифры. Например:

```
char v1[] = "a\xah\129";   //6 char: 'a' '\xa' 'h' '\12' '9' '\0'
char v2[] = "a\xah\127";   //5 char: 'a  '\xa'  'h' '\127' '\0'
char v3[] = "a\xad\127";   //4 char: 'a' '\xad' '\127 '\0'
char v4[] = "a\xad\0127";  //5 char: 'a' '\xad' '\012' '7' '\0'
```

С.3.3. Расширенные символьные наборы

Программа на C++ может писаться и представляться пользователю в рамках символьных наборов, более широких, чем набор из 127 символов ASCII. Там, где реализация поддерживает расширенные наборы символов, идентификаторы, комментарии, символьные константы и строки могут содержать символы вроде α, β и т.д. Для переносимости реализация должна также отображать эти символы в кодировку, содержащую лишь доступные для всех пользователей C++ символы. Так как такой перевод символов происходит до начала компиляции, то он не влияет на семантику программы.

Стандартное кодирование символов из расширенного набора символами менее широкого набора, поддерживаемое в C++, представляется последовательностью четырех или восьми шестнадцатеричных цифр:

universal-character-name:
  \U *XXXXXXXX*
  \u *XXXX*

Здесь *X* соответствует одной шестнадцатеричной цифре. Например, ***u1e2b***. Более короткое обозначение ***uXXXX*** равносильно ***U0000XXXX***. Количество шестнадцатеричных цифр, не равное четырем или восьми, является лексической ошибкой.

Программист может непосредственно использовать такое кодирование, однако в первую очередь оно предназначено для реализации, чтобы последняя использовала ее для представления видимого пользователю расширенного набора более узким внутренним набором символов.

Если вы полагаетесь на специальные среды для представления идентификаторов символами расширенного набора, то вы, тем самым, снижаете переносимость программы. Такую программу труднее читать и понимать, если нет понимания использованного для идентификаторов и комментариев естественного языка. Следовательно, чтобы программа могла восприниматься программистами разных стран, лучше ограничиться английским и символами ASCII.

C.3.4. Знаковые и беззнаковые символы

В зависимости от реализации тип ***char*** может быть знаковым или беззнаковым. Это открывает дверь для неприятных сюрпризов. Например:

```
char c = 255;    // 255 - "все единицы", шестнадцатеричное 0xFF
int i = c;
```

Каково будет значение ***i***? К сожалению, ответ не определен. Во всех известных мне реализациях он зависит от смысла преобразования к типу ***int*** «всех единиц» в ***char***. На платформе SGI Challenge тип ***char*** беззнаковый, так что ответ будет ***255***. На аппаратных платформах Sun SPARC или IBM PC тип ***char*** знаковый и ответ равен ***-1***. В последнем случае компилятор может предупредить о преобразовании литерала ***255*** в значение ***-1*** для типа ***char***. Однако C++ не имеет общего механизма для выявления подобного рода проблем. Чтобы избежать этой проблемы, можно вместо char указывать точно ***unsigned char*** или ***signed char***. К сожалению, некоторые библиотечные функции, такие как ***strcmp()***, принимают только тип ***char*** (§20.4.1).

Тип ***char*** должен вести себя идентично ***signed char*** или ***unsigned char***. Тем не менее, все три типа считаются различными, так что нельзя смешивать указатели на разные ***char***:

```
void f(char c, signed char sc, unsigned char uc)
{
  char* pc = &uc;            // error: нет преобразования указателей
  signed char* psc = pc;     // error: нет преобразования указателей
  unsigned char* puc = pc;   // error: нет преобразования указателей
  psc = puc;                 // error: нет преобразования указателей
}
```

В то же время переменные этих типов можно свободно присваивать друг другу. Однако присваивание слишком больших значений переменной ***signed char*** (§C.6.2.1) по-прежнему не определено. Например:

```
void f(char c, signed char sc, unsigned char uc)
{
  c = 255;      // зависит от реализации
  c = sc;       // ok
  c = uc;       // зависит от реализации
  sc = uc;      // зависит от реализации
  uc = sc;      // ok: преобразование к unsigned
  sc = c;       // зависит от реализации
  uc = c;       // ok: преобразование к unsigned
}
```

Ни одна из этих потенциальных проблем не возникает, если использовать просто ***char***.

С.4. Типы целых литералов

В общем случае, тип целого литерала зависит от его формы, значения и суффикса:

- Если литерал десятичный и не имеет суффикса, то он относится к первому из следующих типов, достаточных для представления его значения: ***int***, ***long int***, ***unsigned long int***.
- Если литерал восьмеричный и не имеет суффикса, то он относится к первому из следующих типов, достаточных для представления его значения: ***int***, ***unsigned int***, ***long int***, ***unsigned long int***.
- Если литерал имеет суффикс ***u*** или ***U***, то его тип — первый из следующих двух типов, достаточных для представления его значения: ***unsigned int***, ***unsigned long int***.
- Если литерал имеет суффикс ***l*** или ***L***, то его тип — первый из следующих двух типов, достаточных для представления его значения: ***long int***, ***unsigned long int***.
- Если литерал имеет суффикс ***ul***, ***lu***, ***uL***, ***Lu***, ***Ul***, ***lU***, ***UL*** или ***LU***, то его тип — ***unsigned long int***.

Например, число ***100000*** имеет тип int на машине с 32-битовыми ***int***, но тип ***long int*** на машине с 16-битовыми int и 32-битовыми ***long***. Аналогично, ***0XA000*** относится к типу ***int*** на машине с 32-битовыми ***int***, но к типу ***unsigned int*** на машине с 16-битовыми ***int***. Таких зависимостей от реализации можно избежать, если явно использовать суффиксы: ***100000L*** всегда (на всех машинах) имеет тип ***long int***, а ***0XA000U*** — тип ***unsigned int*** (на всех машинах).

С.5. Константные выражения

Язык C++ требует *константных выражений* (*constant expressions*) для определения границ массивов (§5.2), меток оператора ***case*** (§6.3.2) и инициализаторов перечислений (§4.8). Вычисление константного выражения приводит к интегральной константе или константе перечисления. Такие выражения состоят из литералов (§4.3.1, §4.4.1, §4.5.1), элементов перечислений (§4.8) и констант, инициализируемых константными выражениями. В шаблонах также можно использовать целый параметр шаблона (§C.13.3). Литералы с плавающей точкой (§4.5.1) можно использовать, преобразовав их явным образом в интегральный тип. Функции, классовые объекты, указатели и ссылки можно использовать в качестве операндов операции ***sizeof*** (§6.2).

С интуитивной точки зрения, константные выражения — это просто выражения, которые компилятор может вычислить до начала компоновки программы (§9.1) и ее запуска на выполнение.

С.6. Неявное преобразование типов

Интегральные типы и типы с плавающей запятой (§4.1.1) можно свободно смешивать в арифметических выражениях и операциях присваивания. По возможности значения преобразуются таким образом, чтобы не терять информацию. К сожалению, преобразования с потерей значений также могут выполняться неявно. В на-

стоящем разделе приведены правила преобразований, связанные с этим проблемы и их решения.

С.6.1. Продвижения (promotions)

Неявные преобразования, сохраняющие значения, называют *продвижениями* (*promotions*). Перед выполнением арифметической операции выполняется *интегральное продвижение* (*integral promotion*) с целью получения значений типа ***int*** из более коротких целых типов. Заметьте, что эти продвижения не продолжаются в сторону ***long*** (если только не задействован тип ***wchar_t*** или элемент перечисления со значением, слишком большим для ***int***). Это соответствует исходной цели таких продвижений в языке С: привести операнды к естественным размерам для арифметических операций.

Интегральные продвижения таковы:

- ***char***, ***signed char***, ***unsigned char***, ***short int*** или ***unsigned short int*** преобразуются в ***int***, если ***int*** может отобразить все значения исходного типа; в противном случае они преобразуются в ***unsigned int***.
- ***wchar_t*** (§4.3) или перечисления (§4.8) преобразуются к первому из перечисленных ниже типов, способному отобразить все их значения: ***int***, ***unsigned int***, ***long*** или ***unsigned long***.
- Битовые поля (§С.8.1) преобразуются в ***int***, если ***int*** может представить все значения битового поля; в противном случае осуществляется преобразование в ***unsigned int***, если этот тип может представить все значения битового поля. В противном случае никакого интегрального продвижения не выполняется вообще.
- Тип ***bool*** преобразуется в ***int***; ***false*** становится нулем, а ***true*** — единицей.

Продвижения используются как часть обычных арифметических преобразований (§С.6.3).

С.6.2. Преобразования

Фундаментальные типы могут преобразовываться друг в друга огромным количеством способов. По моему мнению, позволено слишком много преобразований. Например:

```
void f(double d)
{
  char c = d;     // внимание: double преобразуется к char
}
```

При написании программ вы должны стремиться избегать неопределенного поведения и преобразований, которые незаметным образом «выбрасывают» информацию. Компилятор может предупредить о многих сомнительных преобразованиях. По счастью, многие компиляторы так и делают.

С.6.2.1. Интегральные преобразования

Целое может преобразовываться в иной целый тип. Значение перечислимого типа может преобразовываться к целому типу.

Если целевой тип беззнаковый, результирующее значение есть просто такой набор битов исходного значения, который помещается в целевом типе (старшие биты при необходимости отбрасываются). Точнее, результат будет наименьшим беззнаковым целым, соответствующим остатку от деления исходного целого на *2* в степени *n*, где *n* — это число битов, применяемых для представления беззнакового целого. Например:

```
unsigned char uc = 1023;    // 1111111111: uc становится 11111111, то есть 255
```

Когда целевой тип знаковый, значение остается неизменным, если оно представимо целевым типом; в противном случае результат зависит от реализации:

```
signed char sc = 1023;      // зависит от реализации
```

Возможны следующие результаты: *127* и *−1* (§С.3.4).

Логическое значение или значение перечислимого типа могут неявно преобразовываться в свой целочисленный эквивалент (§4.2, §4.8).

С.6.2.2. Преобразования чисел с плавающей запятой

Значение с плавающей запятой может быть преобразовано в иной тип с плавающей запятой. Если исходное значение точно представимо в целевом типе, результат совпадает с исходным числовым значением. Если исходное значение располагается между двумя соседними значениями целевого типа, результат будет одним из этих значений. В противном случае результат не определен. Например:

```
float f= FLT_MAX;           // наибольшее значение типа float
double d=f;                 // ok:d==f
float f2 = d;               // ok:f2 ==f
double d3 = DBL_MAX;        // наибольшее значение типа double
float f3 = d3;              // не определено, если FLT_MAX<DBL_MAX
```

С.6.2.3. Преобразования указателей и ссылок

Указатель на любой тип объекта может неявно преобразовываться в тип *void** (§5.6). Указатель (ссылка) на производный класс может неявно преобразовываться в тип указателя (ссылки) на доступный и недвусмысленно определяемый базовый класс (§12.2). Отметим, что указатель на функцию или указатель на член класса не могут неявно преобразовываться в тип *void**.

Константное выражение (§С.5), значением которого служит *0*, может неявно преобразовываться в любой указательный тип или тип указателя на член класса (§5.1.1). Например:

```
int* p =
         !   !   !   !   !      !
         ! !  !  !   !   !      !
        !  ! !   !   !   !      !
       !   !    !!!!!!  !!!!!   !!!!1;
```

Тип *T** может неявно преобразовываться в const *T** (§5.4.1). Аналогично, *T&* может неявно преобразовываться в const *T&*.

С.6.2.4. Преобразования указателей на члены классов

Указатели и ссылки на члены классов могут неявно преобразовываться так, как описано в §15.5.1.

С.6.2.5. Преобразования в логический тип

Указатели, значения интегральных типов и типов с плавающей запятой могут неявно преобразовываться в тип ***bool*** (§4.2). Ненулевые значения преобразовываются в ***true***, а нулевые — в ***false***. Например:

```
void f(int* p, int i)
{
  bool is_not_zero = p;    // true если p!=0
  bool b2 = i;             // true если i!=0
}
```

С.6.2.6. Преобразования «значение интегрального типа — значение с плавающей запятой»

Когда значение с плавающей запятой преобразуется в целое значение, его дробная часть отбрасывается. Другими словами, преобразование из числа с плавающей запятой в целое урезает исходное значение. Например, значение ***int(1.6)*** равно ***1***. Если урезанное значение не может быть представлено целевым типом, то результат не определен. Например:

```
int i = 2.7;         // становится 2
char b = 2000.7;     // не определено для 8-битных char
```

Преобразование из целых в типы с плавающей запятой с математической точки зрения корректно настолько, насколько это позволяет аппаратура. Потеря точности происходит в тех случаях, когда целое значение не удается точно представить в виде значения с плавающей запятой. Например, следующее выражение

```
int i = float(1234567890);
```

оставит ***i*** со значением ***1234567936*** на машине с 32-битовыми ***int*** и ***float***.

Ясно, что лучше избегать преобразований, которые потенциально могут исказить исходное значение. Компиляторы могут выявить и предупредить об очевидно наиболее опасных преобразованиях, таких как преобразование числа с плавающей запятой в целое, или ***long int*** в ***char***. Однако в общем случае, выявление всех особенностей преобразований на этапе компиляции недостижимо, так что программисты не должны терять бдительности. Когда одной лишь бдительности недостаточно, можно вставить явные проверки. Например:

```
class check_failed { };

char checked(int i)
{
  char c = i;              // warning: not portable (§C.6.2.1)
  if(i != c) throw check_failed();
  return c;
}

void my_code(int i)
{
```

```
    char c = checked(i);
    // ...
}
```

Гарантированно контролируемые и переносимые урезания при преобразованиях требуют применения ***numeric_limits*** (§22.2).

С.6.3. Обычные арифметические преобразования

Эти преобразования выполняются над операндами любой бинарной (двухоперандной) операции, чтобы привести их к одному общему типу, который затем используется для типа результата операции:

1. Если один из операндов относится к типу ***long double***, то другой операнд преобразуется к типу ***long double***.
   - В противном случае, если один из операндов относится к типу ***double***, то другой операнд преобразуется в ***double***.
   - В противном случае, если один из операндов имеет тип ***float***, то другой преобразуется к типу ***float***.
   - В противном случае над обоими операндами выполняются интегральные продвижения (§С.6.1).
2. Затем, если один из операндов относится к типу ***unsigned long***, то другой операнд преобразуется к ***unsigned long***.
   - В противном случае, если один из операндов относится к типу ***long int***, а другой — к типу ***unsigned int***, то если ***long int*** в состоянии представить все значения типа ***unsigned int***, операнд типа ***unsigned int*** преобразуется в ***long int***; иначе оба операнда преобразуются к типу ***unsigned long int***.
   - В противном случае, если один из операндов относится к типу ***long***, то другой операнд преобразуется в ***long***.
   - В противном случае, если один из операндов относится к типу ***unsigned***, то другой операнд преобразуется в ***unsigned***.
   - В противном случае оба операнда ***int***.

С.7. Многомерные массивы

Нередко нам требуются векторы векторов, или векторы векторов из векторов и т.д. Как такие многомерные массивы представляются в С++? Для ответа на вопрос я сначала покажу, как для этого применить стандартный класс ***vector***. Затем я рассмотрю применение лишь встроенных средств языков С и С++ для построения многомерных массивов.

С.7.1. Векторы

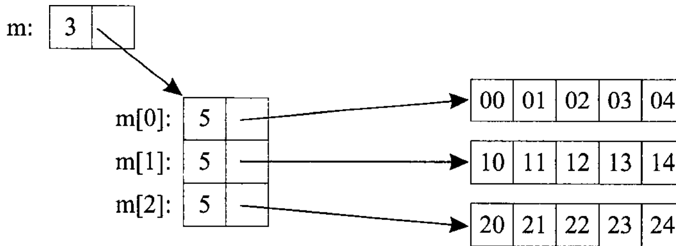
Общее решение можно построить на базе стандартного класса ***vector*** (§16.3):

```
vector<vector<int> > m(3, vector<int>(5));
```

Здесь создается вектор из трех векторов, каждый из которых содержит по 5 целочисленных элементов. Все 15 этих элементов получают умолчательное значение 0. Новые значения элементам можно присвоить следующим образом:

```
void print_m ()
{
 for (int i = 0; i<m.size() ; i++)
  {
   for (int j = 0; j<m[i] .size() ; j++) m[i] [j] = 10*i+j;
  }
}
```

или графически:



Каждый объект, сгенерированный из шаблона ***vector***, содержит число элементов и указатель на элементы, которые в типичном случае содержатся в массиве. Для иллюстрации я задал каждому целому элементу значение, соответствующее его «координатам».

Доступ к элементам осуществляется двойной индексацией. Например, ***m[i] [j]*** — это ***j***-ый элемент ***i***-го вектора. Мы можем осуществить вывод ***m*** следующим образом:

```
void print_m ()
{
 for (int i = 0; i<m.size() ; i++)
  {
   for (int j = 0; j<m[i] .size() ; j++) cout << m[i] [j] << '\t' ;
   cout << '\n' ;
  }
}
```

что приведет к следующему результату:

```
0   1   2   3   4
10  11  12  13  14
20  21  22  23  24
```

Отметим, что ***m*** — это объект типа ***vector***, содержащий другие ***vector***, а вовсе не простой многомерный массив. Поэтому, в частности, можно осуществить изменение размера (§16.3.8) элемента. Например:

```
void init_ma (int ns)
{
 for (int i = 0; i<m.size() ; i++) m[i] .resize(ns) ;
}
```

Не требуется, чтобы вектора типа ***vector<int>***, содержащиеся в ***vector<vector<int> >***, имели одинаковый размер.

С.7.2. Массивы

Встроенные массивы служат источником многих ошибок в программах — особенно в случае многомерных массивов. А новичков они могут сильно смутить. По-возможности им лучше пользоваться типами ***vector***, ***valarray***, ***string*** и т.д.

Многомерные массивы есть массивы массивов; массив 3-на-5 объявляется так:

```
int ma[3] [5] ;    // 3 массива с 5 int в каждом
```

Инициализировать ***ma*** можно следующим образом:

```
void init_ma()
{
 for(int i = 0; i<3; i++)
  {
   for(int j = 0; j<5; j++) ma[i] [j] = 10*i+j;
  }
}
```

или графически:

ma:	00	01	02	03	04	10	11	12	13	14	20	21	22	23	24

Массив ***ma*** — это просто 15 целых чисел, к которым мы обращаемся так, как будто это 3 массива по 5 чисел в каждом. Кроме того, в памяти машины никакого объекта ***ma***, представляющего массив нет — в памяти хранятся лишь элементы. Размеры 3 и 5 существуют лишь в исходном коде. При написании кода мы сами следим за этими размерами. Например, мы можем следующим образом выполнить вывод ***ma***:

```
void print_ma()
{
 for(int i = 0; i<3; i++)
  {
   for(int j = 0; j<5; j++) cout << ma[i] [j] << '\t';
   cout << '\n';
  }
}
```

В языке С++ нельзя использовать запятую для отделения размеров массивов друг от друга, ибо запятая — это операция следования (§6.2.2). В целом, большинство ошибок с массивами отлавливаются компилятором. Например:

```
int bad[3,5];           // error: запятая в константных выражениях не допускается
int good[3] [5];        // 3 массива по 5 int в каждом
int ouch = good[1,4]; // error: good[l,4] это good[4]; int инициализируется типом int*
int nice = good[1] [4];
```

С.7.3. Передача многомерных массивов в функции

Рассмотрим функцию, которая должна манипулировать двумерной матрицей. Если размеры матрицы известны на момент компиляции, то никаких проблем нет:

```
void print_m35 (int m [3] [5] )
{
 for (int i = 0; i<3 ; i++)
  {
   for (int j = 0; j<5; j++) cout << m [i] [j] << '\t' ;
   cout << '\n' ;
  }
}
```

Матрица, представленная как многомерный массив, передается с помощью указателя (а не копируется; §5.3). Размер матрицы по первому измерению не имеет отношения к проблеме нахождения местоположения элемента; он просто сообщает, сколько элементов (здесь 3) и какого типа (здесь — типа ***int***[*5*]) присутствует. Например, посмотрев на предыдущее определение ***ma***, заметим, что зная лишь размер 5 по второму измерению, мы может определить местоположение элемента ***ma***[***i***] [*5*] для любого ***i***. Поэтому размер по первой размерности можно передать в качестве аргумента функции:

```
void print_mi5 (int m [ ] [5] , int dim1)
{
 for (int i = 0; i<dim1; i++)
  {
   for (int j = 0; j<5; j++) cout << m [i] [j] << '\t' ;
   cout << '\n' ;
  }
}
```

Более сложной является ситуация, когда нужно передавать оба размера. Очевидное «решение» не работает:

```
void print_mij (int m [ ] [ ] , int dim1, int dim2)
{
 for (int i = 0; i<dim1; i++)
  {
   for (int j = 0; j<dim2; j++) cout << m [i] [j] << '\t' ;   // сюрприз!
   cout << '\n';
  }
}
```

Во-первых, объявление аргумента в виде ***m***[ ] [ ] незаконно, ибо для нахождения элемента массива нужно знать его вторую размерность. Во-вторых, выражение ***m***[***i***] [***j***] совершенно справедливо интерпретируется как * (* (***m***+***i***) +***j***) , но это не то, что предполагал программист. Правильное решение таково:

```
void print_mij (int* m, int dim1, int dim2)
{
 for (int i = 0; i<dim1; i++)
```

```
  {
    for (int j = 0; j<dim2; j++) cout << m[i*dim2+j] << '\t';    // заковыристо
    cout << '\n';
  }
}
```

Выражение для доступа к элементам в функции ***print_mij***`()` эквивалентно тому, что генерирует компилятор, когда знает вторую размерность.

При вызове этой функции, мы передает матрицу как обычный указатель:

```
int main ()
{
  int v[3][5] = { {0,1,2,3,4}, {10,11,12,13,14}, {20,21,22,23,24} };
  print_m35(v);
  print_mi5(v,3);
  print_mij(&v[0][0],3,5);
}
```

Обратите внимание на `&v[0][0]` в последнем вызове; сработало бы и `v[0]`, так как это равносильно, но для ***v*** будет ошибка типа. Такой тонкий и запутанный код лучше спрятать поглубже. Если вам необходимо работать с многомерными массивами напрямую, попробуйте инкапсулировать имеющие к этому отношение участки кода. Тем самым, вы упростите задачу программисту, который потом будет работать с этой программой. Введение пользовательского типа «многомерный массив» с надлежащей операцией индексации избавит пользователей от хлопот с истинным расположением данных в массиве (§22.4.6).

Стандартный тип ***vector*** (§16.3) такими проблемами не страдает.

С.8. Экономия памяти

При написании нетривиальных программ наступает момент, когда приходится экономить память. Есть два способа «выжимания дополнительной памяти» из имеющейся:

1. Помещать в байт более одного объекта.
2. В разное время использовать одну и ту же память для хранения разных объектов.

Первый способ достигается применением *битовых полей (field)*, а второй способ — применением *объединений (union)*. Мы рассмотрим эти конструкции в последующих разделах. Многие случаи применения битовых полей и объединений относятся к крайним формам оптимизации, основанным на предположениях о конкретных схемах распределения памяти, затрудняющих переносимость программ. Программисту следует дважды подумать перед тем, как применять их. Часто лучше изменить способ управления данными и положиться на динамически выделяемую память (§6.2.6), а не на ее статическое распределение.

С.8.1. Битовые поля

Представляется странным использование целого байта (типы ***char*** или ***bool***) для представления битовых величин (например, переключателя с положениями включено/выключено), но ***char*** — это наименьший объект С++, под который можно выделить адресуемую память (§5.1). Однако можно связать малюсенькие объекты друг с другом в битовые поля в структурах, поля которых определяются по количеству отведенных под них битов. Допускаются неименованные поля. Они не влияют на именованные поля, но помогают расположить последние более эффективно с аппаратной точки зрения:

```
struct PPN                   // физический номер страницы памяти для R6000
{
  unsigned int PFN : 22;     // номер страничного кадра
  int : 3;                   // не используется
  unsigned int CCA : 3;      // алгоритм согласования с кэшем
  bool nonreachable : 1;
  bool dirty : 1;
  bool valid : 1;
  bool global : 1;
};
```

Этот пример иллюстрирует также другое важное применение битовых полей: именование частей заданной извне раскладки памяти. Поля должны иметь интегральный тип или тип перечисления (§4.1.1). Невозможно получить адрес поля. Если не считать такой особенности, то в остальном поля могут использоваться так же, как другие переменные. Отметим, что поле ***bool*** реально представимо единственным битом. В ядре операционной системы или в отладчике тип PPN может использоваться следующим образом:

```
void part_of_VM_system (PPN*p)
{
  // ...
  if(p->dirty)               // содержимое изменилось
  {
    // копирование на диск
    p->dirty = 0;
  }
  // ...
}
```

Как это ни странно, применение битовых полей для упаковки нескольких переменных в одном байте не всегда экономит память. Это лишь экономит память под данные, но объем кода, манипулирующего такими данными, увеличивается на большинстве машин. Известно, что программы сжимаются существенно, когда битовые поля конвертируются в символы. Еще известно, что обращение к битовым полям выполняется медленнее, чем обращение к переменным типа ***int*** или ***char***. Битовые поля — это просто удобное средство для сокрытия применения логических побитовых операций (§6.2.4) для извлечения информации из части машинного слова и для упаковки в часть машинного слова.

С.8.2. Объединения

Объединение (*union*) — это структура, в которой все члены расположены по одному и тому же адресу, так что в памяти машины объединение занимает ровно столько места, сколько требуется для его наибольшего поля. Естественно, в каждый конкретный момент времени объединение может хранить какое-то одно значение. Например, рассмотрим отдельную запись символьной таблицы, содержащую имя и значение:

```
enum Type {S,I};

struct Entry
{
  char* name;
  Type t;
  char* s;    // используем s если t==S
  int i;      // используем i если t==I
};

void f(Entry* p)
{
  if(p->t == S) cout << p->s;
  // ...
}
```

Поскольку членами *s* и *i* нельзя пользоваться одновременно, то выделяемое под них место в памяти тратится впустую. Этого можно избежать, сделав эти переменные членами *объединения* (*union*):

```
union Value
{
  char* s;
  int i;
};
```

В языке C++ нет встроенных средств слежения за тем, какого типа данные хранятся в объединении, так что эта задача ложится на плечи программиста:

```
struct Entry
{
  char* name;
  Type t;
  Value v;   // используем v.s если t==S; используем v.i если t==I
};

void f(Entry* p)
{
  if(p->t == S) cout << p->v.s;
  // ...
}
```

К сожалению, переход на объединение заставляет нас также изменить код, использующий *s*, на *v.s*. Этого можно избежать применением *анонимного объединения* (*anonymous union*), которое не имеет имени и, значит, не является типом. Оно просто гарантирует, что его члены расположены по одному и тому же адресу памяти:

```
struct Entry
{
  char* name;
  Type t;

  union
  {
    char* s;  // используем s если t==S
    int i;    // используем i если t==I
  };
};

void f(Entry* p)
{
  if(p->t == S) cout << p->s;
  // ...
}
```

В результате, код, использующий ***Entry***, не претерпевает изменений.

К сожалению, невозможно гарантировать, что объединение всегда читается корректно, то есть в соответствии с типом записанного в него значения (что порождает тонкие ошибки). Поэтому целесообразно инкапсулировать объединение, что может дать гарантию соответствия между читаемым значением и его типом (§10.6[20]).

Объединения иногда используются неразумно с целью преобразования типов. Обычно это свойственно программистам с опытом работы на языках, не имеющих операций явного преобразования типов и где приходится из-за этого пускаться на всякие хитрости. Следующий пример показывает преобразование из ***int*** в ***int****, основанное на их битовой эквивалентности:

```
union Fudge
{
  int i;
  int* p;
};

int* cheat(int i)
{
  Fudge a;
  a.i = i;
  return a.p;   // плохо
}
```

Это, на самом деле, вовсе и не преобразование: на некоторых машинах ***int*** и ***int**** занимают разный объем памяти, а на других ***int*** не может иметь нечетный адрес. Такое применение объединений опасно и непереносимо, а для преобразования типов существуют явные и переносимые способы (§6.2.7).

Иногда объединения преднамеренно используются для того, чтобы избежать преобразования типов. Например, ***Fudge*** можно применить с тем, чтобы узнать числовое представление нулевого указателя:

```
int main()
{
  Fudge foo;
```

```
    foo.p = 0;
    cout << "the integer value of the pointer 0 is " << foo.i << '\n';
}
```

С.8.3. Объединения и классы

Встречаются нетривиальные объединения, у которых имеются поля с размером, много большим размеров других, часто используемых полей. Поскольку размер объединения совпадает с наибольшим размером его полей, то память расходуется напрасно. Этого можно избежать, применив вместо объединений производные классы.

Объединения не могут иметь поля данных, имеющих тип класса с конструктором, деструктором и операцией присваивания, ибо компилятор не знает, какого типа член объединения ему нужно уничтожать.

С.9. Управление памятью

В C++ имеются три фундаментальных способа использования памяти:

Статическая память, в которую компоновщик помещает объект на все время выполнения программы. В статической памяти располагаются глобальные переменные и переменные из пространств имен, статические члены классов (§10.2.4) и статические локальные переменные функций (§7.1.2). Объект, размещаемый в статической памяти, конструируется один раз и сохраняется до окончания работы программы. У него все время один и тот же адрес. Статические объекты могут вызывать проблемы в многопоточных программах (с разделяемым адресным пространством), поскольку они используются совместно и во избежание проблем требуется их блокировка.

Автоматическая память, в которой располагаются аргументы функций и локальные переменные. При каждом входе в функцию (блок) создается их копия. Такая память создается и уничтожается автоматически — отсюда и ее название. Ее также называют *стековой памятью* (*memory on the stack*). Если вы хотите явно подчеркнуть характер такой памяти, C++ предоставляет для этого необязательное ключевое слово ***auto***.

Свободная память, которую программы запрашивают явным образом для помещения в нее динамически создаваемых объектов, и которую они могут явным образом освободить по исчерпании нужды в этих объектах (*операции **new** и **delete***). Когда программе нужна дополнительная память, она запрашивает ее у операционной системы операцией new. Свободную память называют также *динамической памятью* (*dynamic memory*) или *кучей* (*heap*).

Что касается программиста, то автоматическая и статическая память используются просто, очевидно и неявно. Интерес представляет лишь использование свободной памяти. Хотя выделение памяти (операцией ***new***) и просто, при отсутствии единой и согласованной стратегии освобождения памяти и передачи ее назад менеджеру свободной памяти, вся память может быть исчерпана, что особо характерно для долго работающих программ.

Простейшая стратегия заключается в том, чтобы передать объекты в свободной памяти под управление автоматических объектов. Как следствие, многие контейнеры

именно так и реализуются, то есть управляют элементами, расположенными в свободной памяти (§25.7). Например, автоматический объект типа ***String*** (§11.12) управляет последовательностью символов, расположенных в свободной памяти, и автоматически освобождает память, когда он сам выходит из области видимости. Все стандартные контейнеры (§16.3, глава 17, глава 20, §22.4) могут так реализовываться.

С.9.1. Автоматическая сборка мусора

Когда описанного выше регулярного подхода недостаточно, программист мог бы воспользоваться менеджером памяти, который ищет висячие объекты (на которые никто не ссылается) и уничтожает их (освобождает занятую ими память). Это называют автоматической сборкой мусора или просто *сборкой мусора* (*garbage collection*). Соответствующий менеджер, естественно, называется *сборщиком мусора* (*garbage collector*).

Фундаментальная идея, лежащая в основе сборки мусора, заключается в том, что если в программе нет ссылок на некоторый объект, то этот объект для нее потерян и его самое время уничтожить сборщиком мусора. Например:

```
void f()
{
  int* p = new int;
  p = 0;
  char* q = new char;
}
```

Здесь присваиванием ***p=0*** мы обрываем связь программы и объекта типа ***int***, так что память, выделенную под этот объект можно безболезненно освободить и предоставить для новых объектов. Поэтому последующую переменную типа ***char*** можно расположить в той же области памяти, и указатель ***q*** будет иметь то же самое значение, что было у переменной ***p***.

Стандарт не требует обязательной реализации автоматической сборки мусора, но она явочным порядком внедряется там, где ручная сборка мусора обходится слишком дорого. При сравнении альтернатив принимайте во внимание производительность программы, расход памяти, надежность, переносимость, материальные затраты на программирование, стоимость сборщика мусора и предсказуемость поведения программы.

С.9.1.1. Замаскированные указатели

Что означает, что на объект нет ссылок? Рассмотрим пример:

```
void f()
{
  int* p = new int;
  long i1 = reinterpret_cast<long>(p) & 0xFFFF0000
  long i2 = reinterpret_cast<long>(p) & 0x0000FFFF
  p = 0;

  // точка #1: в этой точке нет указателя на int

  p = reinterpret_cast<int*>(i1|i2) ;
  // теперь int снова адресуется указателем
}
```

Указатели, хранящиеся в программе в переменных иных типов, называются «замаскированными указателями». В частности, указатель, первоначально хранившийся в переменной *p*, был замаскирован в целых переменных *i1* и *i2*. Но сборщику мусора нет дела до замаскированных указателей; дойдя до точки *#1*, он преспокойно освободит память, выделенную ранее под целое значение. Фактически, подобные программы вряд ли будут корректно работать и без сборщика мусора, поскольку применение *reinterpret_cast* для преобразования целых в указатели в лучшем случае зависит от реализации.

Объединение, которое содержит как указатели, так и не указатели, представляет особую проблему для сборщика мусора, поскольку нет способа узнать, содержит объединение в данный момент указатель или нет. Например:

```
union U
{
  int* p;
  int i;
};

void f(U u, U u2, U u3)
{
  u.p = new int;
  u2.i = 999999;
  u.i = 8;
  //...
}
```

Здесь будет надежнее полагать, что любое значение в таком объединении представляет собой указатель. Но умный сборщик мусора может сделать кое-что и получше: он может заметить, что для данной реализации, например, целые не располагаются по нечетным адресам, или что никакой объект не может иметь адрес *8* и т.д. Это позволит сборщику мусора воздержаться от предположения, что объекты со значениями *999999* или *8* используются функцией *f*().

С.9.1.2. Операция delete

Если в конкретной реализации применяется автоматическая сборка мусора, то операции *delete* и *delete*[] больше не нужны, и программист может просто воздержаться от их применения. Однако помимо освобождения памяти эти операции вызывают деструкторы объектов. Поэтому в случае присутствия сборщика мусора,

```
delete p;
```

вызывает деструктор для объекта, на который указывает *p*, а освобождение памяти откладывается до момента ее обработки сборщиком мусора. Одномоментная «переработка» сборщиком мусора множества объектов помогает избежать сильной дефрагментации памяти (§С.9.1.4). Кроме того, нивелируются в общем случае серьезные последствия ошибочного повторного удаления объекта, деструктор которого сам освобождает память.

Как всегда, доступ к объекту после его уничтожения не определен.

С.9.1.3. Деструкторы

В момент, когда сборщик мусора собирается обработать висящий объект, возникает альтернатива:

1. Вызвать для объекта деструктор (если он имеется).
2. Отнестись к объекту просто как к «сырой» памяти (то есть не вызывать деструктор).

По умолчанию сборщик мусора должен выбрать второй вариант, ибо объекты, созданные операцией ***new***, для которых ***delete*** не вызывался, не должны уничтожаться. В то же время, можно так спроектировать сборщик мусора, чтобы он вызывал деструкторы для зарегистрированных у него объектов. Заметьте, что важно уничтожать объекты в правильном порядке, чтобы избегать ситуаций, когда деструктор одного объекта ссылается на объект, который был уничтожен ранее. Сборщику мусора трудно реализовать правильный порядок уничтожения объектов без помощи со стороны программиста.

С.9.1.4. Фрагментация памяти

Когда память выделяется для множества разных объектов, а затем освобождается, то она с неизбежностью *фрагментируется* (*fragments*) — большая часть памяти разбивается на кусочки, слишком малые для эффективного использования. Причина заключается в том, что универсальный распределитель памяти не всегда может найти кусок памяти нужного размера. А применить для объекта кусок памяти несколько большего размера, значит оставить часть памяти неиспользуемой. Поработает программа с таким неэффективным распределителем памяти чуть подольше, и уже добрая половина памяти использована под маленькие фрагменты, которые трудно снова пустить в дело.

Существует несколько методик борьбы с фрагментацией памяти. Простейшая методика заключается в запросе у распределителя (аллокатора, allocator) большого куска памяти, который затем нарезается на одинаковые доли под объекты одинакового размера (§15.3, §19.4.2). Поскольку большинство актов выделения/освобождения памяти относятся к маленьким объектам вроде узлов дерева, объектов связи (links) и т.д., то такая методика оказывается очень эффективной. Иногда аллокаторы могут автоматически применять похожие методики. В любом случае, фрагментация снижается еще сильнее, если все большие «куски» (chunks) имеют одинаковый размер (скажем, размер страницы памяти), так что их самих можно динамически размещать в памяти без фрагментации последней.

Существуют сборщики мусора двух стилей:

1. *Копирующий сборщик* (*copying collector*) для дефрагментации памяти перемещает в ней объекты.
2. *Консервативный сборщик* (*conservative collector*) размещает объекты так, чтобы минимизировать фрагментацию.

С точки зрения языка C++ предпочтительнее консервативный сборщик, ибо исключительно трудно (если вообще возможно) переместить объект в памяти и корректно модифицировать все указатели на него. Кроме того, консервативный сборщик позволяет сосуществовать фрагментам программы на C++ с фрагментами на других языках, например, на C. Традиционно копирующие сборщики пользовались популярностью у программистов, имевших дело с языками типа Lisp и Smalltalk, в которых объекты адресуются исключительно косвенно через уникальные указате-

ли или ссылки. Для большинства больших программ, где важны объем копирования и степень взаимодействия аллокаторов со страничными подсистемами операционных систем, современные консервативные сборщики почти также эффективны, как и копирующие. Для программ поменьше часто достижим практически идеальный вариант, когда сборщик мусора вообще никогда не вызывается — особенно в C++, где многие объекты автоматические.

C.10. Пространства имен

Этот раздел посвящен небольшим подробностям, относящимся к пространствам имен, которые выглядят как мелкие технические детали, но часто мелькают в публичных дискуссиях и реальном программном коде.

C.10.1. Удобство против безопасности

Объявление *using* вносит имя в локальную область видимости, а директива *using* — нет; она просто делает доступными все имена из области видимости, в которой они были объявлены. Например:

```
namespace X
{
  int i,j,k;
}

int k;

void f1 ()
{
  int i = 0;
  using namespace X;      // делает доступными имена из X
  i++;                    // локальная i
  j++;                    // X::j
  k++;                    // error: X::k или глобальная k?
  ::k++;                  // глобальная k
  X::k++;                 // X::k
}

void f2 ()
{
  int i = 0;

  using X::i;             // error: i дважды определяется в f2()
  using X::j;
  using X::k;             // скрывает глобальную k

  i++;
  j++;                    // X::j
  k++;                    // X::k
}
```

Локально объявленное имя (обычным образом или *using*-объявлением) скрывает нелокальные объявления того же имени, а все незаконные перегрузки имени выявляются в точке объявления.

Обратите внимание на ошибку неоднозначности для *k*++ в *f1*(). Глобальные имена не имеют приоритета перед именами из пространства имен, ставших доступными в глобальной области видимости. Это обеспечивает надежную защиту от случайных конфликтов имен и — что важно — нивелирует какие-либо преимущества «заселения» именами глобального пространства (с его неминуемым замусориванием).

Когда библиотеки, объявляющие много имен, становятся доступными благодаря *using*-директивам, очень важно то, что конфликты неиспользуемых имен не считаются ошибками.

Глобальная область видимости — это просто еще одно пространство имен. Оно отличается лишь тем, что его не нужно явно квалифицировать. То есть ::*k* означает, что *k* нужно искать в глобальном пространстве имен и в пространствах имен, упомянутых в using-директивах, расположенных в глобальной области видимости, в то время как *X*::*k* означает «*k*, объявленное в пространстве имен *X* или в пространствах имен, упомянутых в *using*-директивах в *X*» (§8.2.8).

Я надеюсь, что в новых программах с применением пространств имен я увижу радикальное снижение использования глобального пространства имен по сравнению со старыми, традиционными программами на С и С++. Правила работы с пространствами имен нарочито разрабатывались таким образом, чтобы у ленивого почитателя глобального пространства не было никаких преимуществ перед теми, кто не замусоривает глобальную область видимости.

С.10.2. Вложенные пространства имен

Очевидное применение пространства имен заключается в том, что в него помещают весь набор объявлений и определений:

```
namespace X
{
  // все мои объявления
}
```

В общем случае, список объявлений может включать в себя пространства имен. Так возникают вложенные пространства имен. Это допускается из практических соображений и согласуется с идеей, что конструкции могут быть вложенными всегда, если только нет серьезных возражений против этого. Вот пример:

```
void h () ;

namespace X
{
  void g () ;
  // ...
  namespace Y
  {
    void f () ;
    void ff () ;
    // ...
  }
}
```

Применяются обычные правила видимости и квалификации:

```
void X::Y::ff()
{
 f(); g(); h();
}

void X::g()
{
 f();            // error: нет f() в X
 Y::f();         // ok
}

void h()
{
 f();            // error: нет глобальной f()
 Y::f();         // error: нет глобального Y
 X::f();         // error: нет f() в X
 X::У::f();      // ok
}
```

С.10.3. Пространства имен и классы

Пространство имен — это именованная область видимости. Класс — это тип с определенной областью видимости, которая описывает, как могут объекты класса создаваться и использоваться. Таким образом, пространство имен — это более простая концепция, нежели класс, и было бы идеально определить класс как пространство имен плюс несколько дополнительных средств. Это почти так. Пространство имен открыто (§8.2.9.3), а класс закрыт. Это различие проистекает из соображения, что класс должен определять раскладку объектов, а это лучше делать в одном месте. Кроме того, *using*-объявления и *using*-директивы применимы к классам весьма ограниченным образом (§15.2.2).

Пространства имен предпочтительнее классов, если все, что требуется — это инкапсуляция имен. В таком случае аппарат классов для проверки типов и создания объектов не нужен; достаточно более простой концепции пространств имен.

С.11. Управление режимами доступа

В настоящем разделе приведено некоторое количество технических примеров, иллюстрирующих управление режимами доступа в дополнение к примерам из §15.3.

С.11.1. Доступ к членам класса

Рассмотрим фрагмент кода:

```
class X
{
  int priv;

protected:
  int prot;
```

```
public:
  int publ;
  void m();
};
```

Здесь *X::m*() имеет неограниченный доступ:

```
void X::m()
{
  priv = 1;     // ok
  prot = 2;     // ok
  publ = 3;     // ok
}
```

Члены производного класса имеют доступ к открытым и защищенным членам базового класса (§15.3):

```
class Y: public X
{
  void mderived();
};

void Y::mderived()
{
  priv = 1;     // error: priv - private
  prot = 2;     // ok: prot - protected и mderived() есть функция-член производного класса
  publ = 3;     // ok
}
```

Глобальные функции могут обращаться только к открытым членам:

```
void f(Y*p)
{
  p->priv = 1; // error: priv - private
  p->prot = 2; // error: prot -protected, а f() - не друг и не член X или Y
  p->publ = 3; // ok:
}
```

С.11.2. Доступ к базовым классам

Как и члены класса, базовые классы также могут быть объявлены с ключевыми словами ***private***, ***protected*** и ***public***:

```
class X
{
public:
  int a;
  // ...
};

class Y1: public X { };
class Y2: protected X { };
class Y3: private X { };
```

Поскольку *X* является открытым базовым классом для *Y1*, то любая функция может воспользоваться неявным преобразованием от *Y1** к *X** там, где это нужно. Например:

```
void f(Y1* py1, Y2* py2, Y3* py3)
{
 X* px = py1; // ok: X - открытый базовый класс для Y1
 py1->a = 7;  // ok

 px = py2;    // error: X - защищенный базовый класс для Y2
 py2->a = 7;  // error

 px = py3;    // error: X - закрытый базовый класс для Y3
 py3->a = 7;  // error
}
```

Рассмотрим следующий фрагмент:

```
class Y2: protected X { };
class Z2: public Y2 {void f(Y1*, Y2*, Y3*); };
```

Так как *X* является защищенным базовым классом для *Y2*, только члены и друзья класса *Y2* (а также члены и друзья классов, производных от *Y2*, например *Z2*) могут воспользоваться неявным преобразованием от *Y2** к *X** там, где это нужно, как они могут получить доступ к открытым и защищенным членам класса *X*. Например:

```
void Z2::f(Y1* py1, Y2*py2, Y3* py3)
{
 X* px = py1; // ok: X - открытый базовый класс для Y1
 py1->a = 7;  // ok
 px = this;   // ok: X - защищенный базовый класс для Y2, а Z2 наследует Y2
 a = 7;       // ok
 px = py2;    // error: X - защищенный базовый класс для Y2, а Z2 наследует Y2,
              // но мы не знаем, представляет ли py2 объект Z2,
              // или как Y2::X используется в не-Z2 объектах
 py2->a = 7;  // error: Z2 не знает, как Y2::a используется в не-Z2 объектах
 px = py3;    // error: X - закрытый базовый класс для Y3
 py3->a = 7;  // error
}
```

Наконец рассмотрим

```
class Y3: private X {void f(Y1*, Y2*, Y3*); };
```

Поскольку *X* является закрытым базовым классом для *Y3*, только члены и друзья класса *Y3* могут воспользоваться неявным преобразованием от *Y3** к *X** (где это нужно), как они могут обращаться к открытым и защищенным членам класса *X*. Например:

```
void Y3::f(Y1* py1, Y2* py2, Y3* py3)
{
 X* px = py1; // ok: X - открытый базовый класс для Y1
 py1->a = 7;  // ok
 px = py2;    // error: X - защищенный базовый класс для Y2
 py2->a = 7;  // error
 px = py3;    // ok: X - закрытый базовый класс для Y3,
              // и Y3::f() - член класса Y3
 py3->a = 7;  // ok
}
```

С.11.3. Доступ ко вложенным классам

Члены вложенного класса не имеют никаких специальных прав на доступ к членам объемлющего класса. Аналогично, члены объемлющего класса не имеют никаких особых прав на доступ к членам вложенного класса — всюду работают обычные правила доступа (§10.2.2). Например:

```
class Outer
{
  typedef int T;
  int i;
public:
  int i2;
  static int s;
  class Inner
  {
   int x;
   T y;              // error: Outer::T - закрытый
  public:
   void f(Outer* p, int v) ;
  };
  int g (Inner* p) ;
};
void Outer::Inner::f(Outer* p, int v)
{
  p->i = v;          // error: Outer::i - закрытый
  p->i2 = v,         // ok: Outer::i2 - открытый
}
int Outer::g (Inner* p)
{
  p->f(this,2) ; // ok: Inner::f() - открытый
  return p->x;   // error: Inner::x - закрытый
}
```

Часто, однако, бывает полезным предоставить членам вложенного класса неограниченный доступ к членам объемлющего класса. Для этого достаточно объявить вложенный класс другом объемлющего класса. Например:

```
class Outer
{
  typedef int T;
  int i;
public:
  class Inner;     // предварительное объявление класса
  friend class Inner;
  class Inner
  {
   int x;
   T y;            // ok: Inner - друг
```

```
    public:
      void f(Outer* p, int v);
    };
};

void Outer::Inner::f(Outer* p, int v)
{
  p->i = v;         // ok: Inner - друг
}
```

С.11.4. Отношение «дружбы»

Отношение дружбы не транзитивно и не наследуемо. Например:

```
class A
{
  friend class B;
  int a;
};

class B
{
  friend class C;
};

class C
{
  void f(A* p)
  {
    p->a++;         // error: C - не друг A, а друг друга A
  }
};

class D: public B
{
  void f(A* p)
  {
    p->a++;         //error: D - не друг A, а наследует от друга A
  }
};
```

С.12. Указатели на члены классов

Естественно, что понятие указателя на члены класса (§15.5) применяется как к членам данных, так и к функциям-членам, имеющим аргументы и возвращаемые значения. Например:

```
struct C
{
  const char* val;
  int i;
  void print(int x) {cout << val << x << '\n';}
  int fl(int);
```

```
  void f2();
  C(const char* v) {val = v;}
};

typedef void (C::*PMFI)(int); // указатель на функцию-член класса C с аргументом int
typedef const char* C::*PM;   // указатель на поле данных типа const char* класса C

void f(C& z1, C& z2)
{
  C* p = &z2;
  PMFI pf= &C::print;
  PM pm = &C::val;

  z1.print(1);
  (z1.*pf)(2);
  zl.*pm = "nv1";
  p->*pm = "nv2";
  z2.print(3);
  (p->*pf)(4);

  pf = &C::f1;            // error: неправильный тип возврата
  pf = &C::f2;            // error: неправильный тип аргумента
  pm = &C::i;             // error: несоответствие типа
  pm = pf;                // error: несоответствие типа
}
```

Тип указателя на функцию проверяется точно так же, как любой другой тип.

С.13. Шаблоны

Шаблон класса описывает, как можно сгенерировать класс по заданному подходящему набору аргументов шаблона. Аналогично, шаблон функции описывает, как можно сгенерировать функцию по заданному подходящему набору аргументов шаблона. Таким образом, шаблоны используются для генерирования типов и исполнимого кода. Вместе с такой выразительной мощью возникают и некоторые сложности, связанные по большей части со множеством контекстов определения и использования шаблонов.

С.13.1. Статические члены

Шаблон класса может иметь статические члены. Каждый класс, сгенерированный из шаблона, имеет свою собственную копию статических членов. Статические члены должны определяться отдельно и могут иметь специализации. Например:

```
template<class T> class X
{
  // ...
  static T def_val;
  static T* new_X(T a = def_val);
};
```

```
template<class T> T X<T>::def_val;        // инициализируется X<T>()
template<class T> T* X<T>::new_X(T a) {/*...*/}

template<> int X<int>::def_val = 0;
template<> int* X<int>::new_X(int i) {/*...*/}
```

Если вы хотите сделать объект или функцию общими для всех членов любого сгенерированного из шаблона класса, вы можете разместить их в нешаблонном базовом классе. Например:

```
struct B
{
  static B* nil;  // общий нулевой указатель для всех классов, производных от B
};

template<class T> class X: public B
{
  // ...
};

B* B::nil = 0;
```

С.13.2. Друзья

Шаблоны классов могут иметь друзей. Рассмотрим, например, ***Matrix*** и ***Vector*** из §11.5. В общем случае они будут представляться шаблонами:

```
template<class T> class Matrix;

template<class T> class Vector
{
  T v[4];
public:
  friend Vector operator* (const Matrix<T>&, const Vector&);
  // ...
};

template<class T> class Matrix
{
  Vector<T> v[4];
public:
  friend Vector<T> operator* (const Matrix&, const Vector<T>&);
  // ...
};
```

Затем можно определить операцию умножения, имеющую прямой доступ к данным из ***Matrix*** и ***Vector***:

```
template<class T> Vector<T> operator* (const Matrix<T>& m, const Vector<T>& v)
{
  // ... используем m.v[i] и v.v[i] для прямого доступа к элементам
}
```

Друзья не оказывают влияния на область видимости, в которой определен шаблонный класс; также они не влияют на область видимости, в которой этот шаблон

используется. Функции-друзья и операции-друзья находятся путем поиска, базирующегося на типе их аргументов (§11.2.4, §11.5.1). Как и функции-члены, функции-друзья конкретизируются (§C.13.9.1) только, если осуществляется их вызов.

C.13.3. Шаблоны в качестве параметров шаблонов

Бывает полезным передать шаблон (а не класс или объект) в качестве аргумента шаблона. Например:

```
template<class T, template<class> class C> class Xrefd
{
  C<T> mems;
  C<T*> refs;
  // ...
};

Xrefd<Entry, vector> x1;  // хранит перекрестные ссылки для Entry в контейнере типа vector
Xrefd<Record, set> x2;    // хранит перекрестные ссылки для Record в контейнере типа set
```

Чтобы объявить шаблон как параметр шаблона, нужно указать требующиеся ему аргументы. Например, мы объявляем параметр C шаблона *Xrefd* как шаблонный класс, которому требуется один аргумент-тип. Если мы этого не сделали бы, то не смогли бы использовать специализации C. Причина нашего стремления использовать шаблон в качестве параметра шаблона заключается в том, что мы хотели бы конкретизировать его разными типами аргументов (такими как *T* и *T** в предыдущем примере). То есть мы хотели бы выражать объявления членов шаблона в терминах другого шаблона, который был бы параметром и задавался пользователем.

Когда шаблону требуется контейнер для хранения элементов с типом аргумента шаблона, лучше передать ему тип контейнера (§13.6, §17.3.1).

Только классовые шаблоны могут выступать в роли аргументов другого шаблона.

C.13.4. Логический вывод аргументов функциональных шаблонов

Компилятор может осуществить логический вывод типа аргумента шаблона, *T* или *TT*, и нетипового аргумента шаблона — *I*, исходя из аргумента шаблона функции с типом, составленным из следующих конструкций:

```
T                          const T                       volatile T
T*                         T&                            T[constant_expression]
type[I]                    class_template_name<T>        class_template name<I>
TT<T>                      T<I>                          T<>
T type::*                  T T::*                        type T::*
T (*)(args)                type (T::*)(args)             T (type::*)(args)
type (type::*)(args_TI)    T (T::*)(args_TI)             type (T::*)(args_TI)
T (type::*)(args_TI)       type (*)(args_TI)
```

Здесь *args_TI* — это список параметров, по которому можно определить *T* или *I* согласно рекурсивной процедуре применения этих правил, а *args* — это список параметров, не допускающий логического вывода. Если таким образом можно вывести не все параметры, вызов считается неоднозначным. Например:

```
template<class T, class U> void f(const T*, U(*)(U));

int g(int);

void h(const char* p)
{
 f(p,g);    // T есть char, U есть int
 f(p,h);    // error: невозможно вывести U
}
```

Глядя на аргументы первого вызова *f*(), мы легко выводим все аргументы функционального шаблона. Для второго же вызова *f*() мы видим, что *h*() не соответствует образцу *U*(*)(*U*), поскольку тип аргумента функции *h*() отличается от типа возвращаемого ею значения.

Если параметр шаблона выводится из более чем одного аргумента функции, результатом каждого вывода должен быть один и тот же тип. В противном случае вызов считается ошибочным. Например:

```
template<class T> void f(T i, T* p);

void g(int i)
{
 f(i,&i)                // ok
 f(i,"Remember!");      // error, неоднозначность: T есть int, или T есть const char?
}
```

С.13.5. Шаблоны и ключевое слово typename

Чтобы сделать обобщенное программирование более простым и более общим, контейнеры стандартной библиотеки предоставляют набор стандартных функций и типов (§16.3.1). Например:

```
template<class T> class vector
{
public:
  typedef T* iterator;

  iterator begin();
  iterator end();
  // ...
};

template<class T> class list
{
  class link {/* ... */};

public:
  typedef link* iterator;

  iterator begin();
  iterator end();
  // ...
};
```

Это искушает нас написать

```
template<class C> void f(C& v)
{
  C::iterator i = v.begin();    // синтаксическая ошибка
  // ...
}
```

К сожалению, от компилятора нельзя требовать слишком многого и он не понимает, что ***C::iterator*** — это имя типа. В некоторых частных случаях «умный» компилятор мог бы распознать, относится имя к типу, или к чему-нибудь еще, что не является типом (к функции или шаблону). Но в общем случае это невозможно. Действительно, рассмотрим пример, лишенный подсказок насчет его смысла:

```
int y;

template<class T> void g(T& v)
{
  T::x(y);        // вызов функции или объявление переменной?
}
```

Является ли здесь ***T::x*** функцией, вызванной с ***y*** в качестве аргумента? Или мы намеревались объявить локальную переменную y с типом ***T::x***, поставив здесь по странным причинам необязательные круглые скобки? В принципе можно представить себе реальные программы, в которых ***X::x(y)*** будет функциональным вызовом, а ***Y::x(y)*** будет объявлением.

Эта неоднозначность разрешается просто: если не указано обратного, то идентификатор относят к чему-то такому, что не является типом или шаблоном. А если мы хотим трактовать идентификатор как имя типа, мы можем это сделать с помощью ключевого слова ***typename***:

```
template<class C> void h(C& v)
{
  typename C::iterator i = v.begin();
  // ...
}
```

Ключевое слово ***typename*** можно поставить перед квалифицированным именем с целью утверждения, что имя относится к типу. В этом отношении оно напоминает ключевые слова ***struct*** и ***class***.

Ключевое слово ***typename*** необходимо, если имя типа зависит от параметров шаблона. Например:

```
template<class T>
void k(vector<T>& v)
{
  vector<T>::iterator i = v.begin();             // error: отсутствует typename
  typename vector <T>::iterator i = v.begin();   // ok
  // ...
}
```

В этом случае компилятор может быть и смог бы определить, что ***iterator*** есть имя типа в каждой конкретизации шаблона ***vector***, но стандартом от него этого не требуется, так как это было бы непереносимым расширением языка. Единственный контекст, в котором компилятор может определить, что имена, зависящие от аргу-

мента шаблона, являются именами типов — это несколько частных случаев, в которых лишь имена типов разрешены грамматикой. Например, в случае спецификаторов, перечисленных в §A.8.1.

Ключевое слово ***typename*** можно также использовать как альтернативу ключевому слову ***class*** в объявлениях шаблонов:

```
template<typename T> void f(T);
```

Поскольку я не слишком быстро работаю с клавиатурой и мне всегда не хватает места на экране, я предпочитаю более короткую запись

```
template<class T> void f(T);
```

С.13.6. Ключевое слово template в качестве квалификатора

Необходимость в квалификаторе ***typename*** возникла потому, что мы можем обращаться к членам классов, которые являются типами, и к членам, которые типами не являются. Аналогично, имеется необходимость отличать шаблонные члены классов от членов, шаблонами не являющихся. Рассмотрим возможный интерфейс к универсальному менеджеру памяти:

```
class Memory
{
public:
  template<class T> T* get_new();
  template<class T> void release(T&);
  // ...
};

template<class Allocator> void f(Allocator& m)
{
  int* p1 = m.get_new<int>();              // error
  int* p2 = m.template get_new<int>();     // явная квалификация
  // ...
  m.release(p1); // параметр шаблона выводится (нет нужды в явной квалификации)
  m.release(p2);
}
```

Явная квалификация при вызове ***get_new***() необходима, поскольку здесь логический вывод параметра невозможен. В этом случае нужно использовать префикс ***template***, чтобы информировать компилятор о том, что имя ***get_new*** относится к шаблонному члену класса и явная квалификация желаемым типом элементов возможна. Без ключевого слова ***template*** будет получено сообщение об ошибке, поскольку знак < будет воспринят как знак операции сравнения «меньше чем». Это редкий случай, поскольку в большинстве случаев удается выполнить логический вывод параметров шаблона.

С.13.7. Конкретизация

Генерация корректного машинного кода из определения шаблона и его использования ложится на плечи реализаций языка C++. Из определения шаблона класса и набора аргументов шаблона компилятор должен сгенерировать определение класса и определения тех его функций-членов, которые находят применение в программе.

Из определения шаблона функции компилятор должен генерировать конкретные функции. Эти процессы называются *конкретизацией шаблона* (*template instantiation*)[1].

Сгенерированные классы и функции называются *специализациями* (*specializations*). Когда нужно различить сгенерированные специализации и специализации, написанные программистом вручную (§13.5), применяют более подробные термины — *сгенерированные специализации* (*generated specializations*) и *явные специализации* (*explicit specializations*), соответственно. Последний термин имеет и другие вариации — *специализации, определяемые пользователем (user-defined specializations)* или просто *специализации.*

Чтобы использовать шаблоны в нетривиальных программах, программист должен понимать, как имена, используемые в определении шаблона, связываются с объявлениями, и как можно организовать исходный код (§13.7).

По умолчанию, компилятор генерирует классы и функции из шаблонов в соответствии с правилами связывания имен (§C.13.8). То есть программист не обязан явно указывать, какие версии каких шаблонов нужно сгенерировать. Это очень важно, потому что программисту бывает трудно понять, какие именно версии шаблонов нужны. Часто в реализациях библиотек используются шаблоны, о которых программист ничего не слышал, а бывает и что знакомые шаблоны используют аргументы неизвестного ему типа. В общем случае, набор функций, которые требуется сгенерировать, можно выявить посредством рекурсивной проверки шаблонов, используемых в необходимых программе библиотеках. Для выполнения такой задачи компьютеры приспособлены лучше человека.

Иногда, все же, программисту бывает важно точно определить, в каких именно местах должен генерироваться код из шаблона (§C.13.10). Тем самым программист получает полный контроль над контекстом конкретизации. Для большинства компиляторов это также означает получение полного контроля над моментом конкретизации. В частности, явная конкретизация может применяться для получения ошибок компиляции в предсказуемое время, а не в тот момент, что выгоден реализации. В определенных случаях важна предсказуемость процесса построения программы.

C.13.8. Связывание имен

Важно определять шаблоны функций так, чтобы они как можно меньше зависели от нелокальной информации. Дело в том, что потом шаблоны будут использоваться для генерации функций и классов, отталкиваясь от неизвестных типов и в неизвестных контекстах. Всякая тонкая зависимость от контекста всплывет все равно в процессе отладки, а программисту не всегда хочется в деталях разбираться в устройстве шаблонов. Универсальное правило — по-возможности избегать глобальных имен — приобретает особую остроту в случае шаблонов. Поэтому мы стремимся сделать определения шаблонов как можно более самодостаточными и передавать то, что в противном случае было бы глобальным контекстом, в форме параметров шаблона (например, свойств (traits); §13.4, §20.2.1).

---

[1] Встречающийся в литературе термин *инстанцирование* труден для произношения. — *Прим. ред.*

Но все же, некоторые нелокальные имена использовать приходится. Например, чаще всего пишут целый набор взаимодействующих друг с другом шаблонов функций, а не единственную самодостаточную функцию. Такие взаимодействующие функции могут (но не обязательно) быть членами классов. Иногда наилучшим выбором становятся нелокальные функции. Типичным примером служат вызовы функций ***swap***() и ***less***() из функции ***sort***() (§13.5.2). Алгоритмы стандартной библиотеки предлагают массу примеров на эту тему (глава 18).

Еще одним источником нелокальных имен в определениях шаблонов служат операции с общепринятыми именами и семантикой, например +, *, [] и ***sort***(). Рассмотрим:

```
#indude <vector>

bool tracing;
//...
template<class T> T sum (std::vector<T>& v)
{
  T t = 0;
  if(tracing) cerr << "sum (" << &v << ") \n";

  for (int i = 0; i<v.size () ; i++) t = t + v[i] ;
  return t;
}
//...
#include <quad.h>

void f(std::vector<Quad>& v)
{
  Quad c = sum (v) ;
}
```

Невинно выглядящая шаблонная функция ***sum***() зависит от операции +. В данном примере операция + определена в <***quad.h***>:

```
Quad operator+ (Quad, Quad) ;
```

Важно, что когда определяется ***sum***(), в области видимости нет никаких чисел ***Quad***, и автор функции ***sum***() может ничего не знать о классе ***Quad***. В частности, операция + может определяться в программе позже функции ***sum***(), и даже позже по времени.

Процесс поиска объявлений имен, явно или неявно используемых в шаблоне, называется *связыванием имен* (*name binding*). Главная проблема со связыванием имен в шаблонах состоит в том, что к конкретизации шаблонов имеют отношение три контекста, четко разделить которые невозможно:

1. Контекст определения шаблона
2. Контекст объявления типов аргументов
3. Контекст использования шаблона

С.13.8.1. Зависимые имена

Определяя шаблон функции, мы хотим быть уверены в том, что имеется достаточный контекст этого определения по отношению к фактическим аргументам без

захватывания «случайного мусора» в точке использования. Помогая нам в этом отношении, язык разделяет все имена, используемые в определении шаблона, на две категории:

1. Имена, зависящие от аргумента шаблона. Такие имена связываются в точке конкретизации (§С.13.8.3). В примере с функцией ***sum***() операция + определяется контекстом конкретизации, поскольку она принимает операнды типа, совпадающего с типом аргументов шаблона.
2. Имена, не зависящие от аргумента шаблона. Эти имена связываются в точке определения шаблона (§С.13.8.2). В примере с функцией ***sum***() шаблон vector определяется в заголовочном файле <***vector***>, и когда компилятор обрабатывает определение ***sum***(), логическая переменная ***tracing*** находится в области видимости.

Простейшим определением того, что «*N* зависит от шаблонного параметра *T*» было бы «*N* является членом *T*». К сожалению, этого недостаточно; сложение чисел типа ***Quad*** (§С.13.8) служит примером обратного. Поэтому говорят, что вызов функции зависит от аргумента шаблона, если и только если выполняется одно из двух следующих условий:

1. Тип фактического аргумента зависит от параметра шаблона *T* согласно правилам логического вывода типа (§13.3.1). Например, ***f***(***T***(***1***)), ***f***(***t***), ***f***(***g***(***t***)) и ***f***(&***t***) в предположении, что ***t*** — это ***T***.
2. Вызываемая функция имеет формальный параметр, зависящий от *T* согласно правилам логического вывода типа (§13.3.1). Например, ***f***(***T***), ***f***(***list***<***T***>&) и ***f***(***const T****).

По сути дела, имя в вызываемой функции зависимо тогда, когда его зависимость очевидна при просмотре аргументов или формальных параметров функции.

Вызов, в котором аргумент случайно соответствует фактическому типу параметра шаблона, не является зависимым. Например:

```
template<class T> T f(T a)
{
  return g(1);   //error: нет g() в области видимости и g(1) не зависит от T
}

int g(int);
int z=f(2);
```

Не имеет значения, что для вызова ***f***(***2***) параметр *T* оказался ***int*** и аргумент ***g***() также имеет тип ***int***. Если бы вызов ***g***(***1***) был зависимым, то его смысл был бы абсолютно непонятен любому, кто читал бы это определение шаблона. Если программист хочет, чтобы вызов связывался с ***g***(***int***), объявление последнего должно располагаться перед определением шаблона, так чтобы оно попало в область видимости в момент анализа шаблона. В принципе, это правило аналогично соответствующему правилу для нешаблонных функций.

Кроме имен функций имена переменных, типов, констант и т.д. могут быть зависимыми, если их тип зависит от параметра шаблона. Например:

```
template<class T> void fct (const T& a)
{
  typename T: :Memtype p = a.p;       // p и Memtype зависят от T
  cout << a.i << ' ' << p->j;         // i и j зависят от T
}
```

С.13.8.2. Связывание в точке определения

Когда компилятор видит определение шаблона, он определяет, какие имена являются зависимыми (§С.13.8.1). Если имя зависимое, поиск его объявления нужно отложить до момента конкретизации (§С.13.8.3).

Имена, не зависящие от аргумента шаблона, должны находиться в области видимости (§4.9.4) в точке определения. Например:

```
int x;

template<class T> T f(T a)
{
  x++;            // ok
  y++;            // error: нет y в области видимости, и y не зависит от T
  return a;
}

int y;
int z = f(2) ;
```

Если объявление находится, то именно оно и используется, несмотря на то, что впоследствии могло быть найдено и объявление «получше». Например:

```
void g (double) ;

template<class T> class X: public T
{
public:
  void f() {g(2) ; }  // вызывается g(double);
  // ...
};

void g (int) ;
class Z { } ;

void h (X<Z> x)
{
  x.f() ;
}
```

Когда генерируется определение для *X<Z>::f()*, *g(int)* не рассматривается, поскольку его объявление расположено позже определения *X*. Не важно, что *X* используется после объявления *g(int)*. Кроме того, вызов, не являющийся зависимым, не может быть отпасован в базовый класс:

```
class Y {public: void g (int) ; } ;

void h (X<Y>x)
{
  x.f() ;
}
```

И снова `X<Y>::f()` вызовет `g(double)`. Если бы программист хотел вызвать `g()` из базового класса *T*, определение `f()` должно было бы иметь вид:

```
template<class T> class XX: public T
{
void f() {T::g(2);}     // вызывается T::g()
 // ...
};
```

Здесь применено эмпирическое правило, гласящее, что «определение шаблона должно быть настолько самодостаточным, насколько это вообще возможно» (§С.13.8).

С.13.8.3. Связывание в точке конкретизации

Каждое применение шаблона для фиксированного набора его аргументов определяет *точку конкретизации* (*point of instantiation*). Эта точка находится в ближайшей охватывающей ее глобальной области видимости или области видимости пространства имен, непосредственно *перед объявлением, использующим шаблон.* Например:

```
struct X { X(int); /* ... */ };

void g(X);

template<class T> void f(T a) {g(a);}

void h()
{
 extern void g(int);
 f(2);                 // вызывается f(X(2)); то есть f<X>(X(2))
}
```

Здесь точка конкретизации для *f* расположена перед `h()`, так что функция `g()`, вызываемая из `f()`, есть глобальная `g(X)`, а не локальная `g(int)`.

Из определения точки конкретизации вытекает, что *параметр шаблона никогда не может быть связан с локальным именем или членом класса.* Например:

```
void f()
{
 struct X {/*...*/};   // локальная структура
 vector<X> v;          // error: нельзя использовать локальную структуру
                       // в качестве параметра шаблона
 // ...
}
```

Кроме того, неквалифицированные имена, применяемые в шаблонах, также не могут быть связаны с локальными именами. И наконец, даже если шаблон впервые используется внутри класса, неквалифицированные имена из определения шаблона не могут быть связаны с членами этого класса. Такое игнорирование локальных имен существенно для предотвращения неприятного поведения, характерного для макросов:

```
template<class T> void sort(vector<T>& v)
{
 sort(v.begin(),v.end());    // используется sort() стандартной биб-ки
```

```
                          // (без явного указания std::)
}

class Container
{
  vector<int> v;

public:
  void sort()
  {
    ::sort(v);      //sort(vector<int>&) вызывает std::sort(), а не Container::sort()
  }
  //....
};
```

Если бы `sort(vector<T>&)` вызывал `sort()`, используя `std::sort()`, результат был бы тем же, а код — был бы прозрачнее.

Если точка конкретизации для шаблона, определенного в пространстве имен, находится в другом пространстве имен, то для связывания доступны имена из обоих пространств имен. Как обычно, для выбора между именами из разных пространств имен применяются правила разрешения перегрузки (§8.2.9.2).

Еще раз отметим, что шаблон, используемый несколько раз с одним и тем же набором аргументов, имеет несколько точек конкретизации. Если при этом связывание независимых имен различно, программа ошибочна. Однако такую ошибку реализациям выявить нелегко, особенно если разные точки конкретизации находятся в разных единицах трансляции. Лучше всего стараться избегать сложностей со связыванием имен, минимизируя применение нелокальных имен в шаблонах и применяя заголовочные файлы для согласования контекстов их использования.

С.13.8.4. Шаблоны и пространства имен

Когда вызывается функция, ее объявление может быть найдено даже вне текущей области видимости при условии, что она объявлена в том же пространстве имен, что и один из ее аргументов (§8.2.6). Это очень важно для функций, вызываемых из определений шаблонов, поскольку при помощи этого механизма находятся зависимые функции в момент конкретизации.

Специализации шаблонов могут генерироваться в любой точке конкретизации (§С.13.8.3), в любой точке, следующей за ней в той же единице трансляции, или в единице трансляции, специально созданной для генерации специализаций. Это отражает три очевидные стратегии, которые могут использоваться в реализациях С++ для генерирования специализаций:

1. Генерировать специализацию сразу, как только первый раз встречается соответствующий вызов.
2. В конце единицы трансляции генерировать все специализации, необходимые для этой единицы трансляции.
3. Когда просмотрены все единицы трансляции, генерировать все специализации, необходимые программе.

Перечисленные стратегии имеют свои преимущества и недостатки; допускаются также и комбинации этих стратегий.

В любом случае, связывание независимых имен производится в точке определения шаблона. Связывание зависимых имен выполняется путем просмотра:

1. Имен в области видимости в точке, где определяется шаблон.
2. Имен из пространства имен аргумента зависимого вызова (глобальные функции рассматриваются в пространстве имен встроенных типов).

Например:

```
namespace N
{
  class A {/*...*/};
  char f(A);
}

char f(int);

template<class T> char g(T t) {return f(t);}
char c = g(N::A());          // приводит к вызову N::f(N::A)
```

Здесь вызов *f(t)* очевидно зависим, так что мы не можем связать *f* с *f(N::A)* или с *f(int)* в точке определения. Чтобы сгенерировать специализацию для *g<N::A>(N::A)*, реализация ищет функции с именем *f()* пространстве имен *N* и находит *N::f(N::A)*.

Программу следует считать неправильной, если можно получить разные результаты, выбрав разные точки конкретизации или разное содержимое пространств имен в разных контекстах генерации специализации. Например:

```
namespace N
{
  class A {/* ...*/};
  char f(A, int);
}

template<class T, class T2> char g(T t, T2 t2) {return f(t,t2);}

char c = g(N::A(), 'a');    // error: возможны разные варианты разрешения f(t)

namespace N                 // добавка к пространству имен N (§8.2.9.3)
{
  void f(A, char);
}
```

Мы могли бы сгенерировать специализацию в точке конкретизации и получить вызов *f(N::A, int)*. Или могли бы подождать и сгенерировать специализацию в конце единицы трансляции и получить вызов *f(N::A, char)*. Следовательно, вызов *g(N::A(), 'a')* является ошибкой.

Вообще-то, это очень неряшливое программирование — вызывать перегруженную функцию между двумя ее объявлениями. В большой программе программист может и не обнаружить подвоха. В нашем конкретном случае компилятор вылавливает неоднозначность. Но если такие вещи встречаются в разных единицах трансляции, то обнаружить проблему будет намного сложнее — реализации не обязаны вылавливать подобные ошибки.

Большинство проблем с неоднозначным разрешением вызовов функций связано со встроенными типами. Поэтому большинство средств лечения проблемы опирается на более осторожное использование аргументов встроенных типов.

Встроенные типы не имеют ассоциированного с ними пространства имен. Следовательно, разрешение зависимых имен не выполняет разрешения перегрузки объявлений, видимых до и после определения шаблона. Например:

```
int f(int) ;

void ff(int) ;
void ff(char) ;

template<class T> T g(T t)  {ff(t) ; return f(t) ; }

char f(char) ;
char c = g('a') ;              //вызывается ff(char);
```

Очевидно, что подобных хитростей лучше избегать.

С.13.9. Когда нужны специализации

Специализацию шаблона класса нужно генерировать лишь тогда, когда требуется определение класса. Например, если используется указатель, то определение класса не нужно. Например:

```
class X;
X* p;          // ok: определения X не требуется
X  a;          // error: требуется определение X
```

Шаблонный класс не конкретизируется до тех пор, пока действительно не потребуется его определение. Например:

```
template<class T> class Link
{
  Link* suc;     // ok: нет нужды в определении Link (пока)
  // ...
};

Link<int>* pl;  // не требуется конкретизации Link<int>
Link<int> lnk;  // здесь уже нужно конкретизировать Link<int>
```

Точка конкретизации находится там, где впервые понадобилось определение.

С.13.9.1. Конкретизация шаблона функции

Реализация конкретизирует шаблон функции только при использовании (вызове) функции. В частности, *конкретизация шаблона класса не влечет за собой конкретизации всех его членов*, и даже членов, полностью определенных в рамках объявления классового шаблона. Это дает программисту определенную гибкость при определении шаблона класса. Например:

```
template<class T> class List
{
  // ...
  void sort() ;
};

class Glob { /* нет операций сравнения */ };
```

```
void f(List<Glob>& lb, List<string>& ls)
{
  ls.sort();
  // ... используем операции над lb, но не lb.sort()
}
```

Здесь конкретизируется функция *List<string>::sort()*, но не функция *List<Glob>::sort()*. Это и уменьшает объем работы, и избавляет нас от необходимости перепроектировать программу. Если бы функция *List<Glob>::sort()* была сгенерирована, нам пришлось бы либо добавить в тип *Glob* операции, требуемые функцией *List::sort()*, переопределить *sort()* так, чтобы она не была членом *List*, или применить какой-либо другой контейнер для объектов типа *Glob*.

С.13.10. Явная конкретизация

Запрос на явную конкретизацию состоит в объявлении специализации после ключевого слова template (за которым не следует символ <):

```
template class vector<int>;                    // класс
template int& vector<int>::operator[](int);   // функция-член
template int convert<int,double>(double);     // функция
```

Объявление шаблона стартует с *template<*, в то время как просто *template* означает начало явного запроса на конкретизацию. Заметьте, что *template* предваряет полное объявление; констатации одного лишь имени не достаточно:

```
template vector<int>::operator[];             // синтаксическая ошибка
template convert<int,double>;                 // синтаксическая ошибка
```

Как и в случае вызовов шаблонных функций, аргументы шаблона, которые выводятся из аргументов функций, можно опустить (§13.3.1). Например:

```
template int convert<int,double>(double);     // ok (избыточно)
template int convert<int>(double);            // ok
```

Когда *явно конкретизируется* шаблон класса, конкретизируются и *все его функции-члены*.

Заметьте, что явную конкретизацию можно применить для проверки ограничений (§13.6.2). Например:

```
template<class T> class Calls_foo
{
  void constraints(T t) {foo(t);}              // call from every constructor
  // ...
};

template class Calls_foo<int>;                 // error: foo(int) не определена
template void Calls_foo<Shape*>::constraints(Shape*); // error: foo(Shape*) не определена
```

Запросы на конкретизацию могут сильно влиять на время компоновки и эффективность повторных компиляций. Я знаю примеры, когда помещение большинства конкретизаций шаблонов в одну единицу трансляции сокращало время компиляции с нескольких часов до нескольких минут.

Иметь два определения для одной и той же специализации — ошибка. Не имеет значения, являются ли они специализациями, определяемыми пользователем (§13.5), неявно сгенерированными (§С.13.7) или явно запрошенными. Однако компилятор не обязан диагностировать множественные конкретизации в разных единицах трансляции. Это позволяет «умным реализациям» игнорировать избыточные конкретизации и, тем самым, избегать проблем, связанных с применением библиотек, использующих явную конкретизацию (§С.13.7). Реализации, однако, не обязаны быть умными. Пользователи «менее умных» реализаций должны сами избегать множественных конкретизаций. Самое плохое, что может случиться в противном случае — это отказ запуска программы.

Язык С++ не требует применения явной конкретизации. Явная конкретизация — это дополнительный механизм для «ручной» оптимизации процессов компиляции и компоновки (§С.13.7).

С.14. Советы

1. Фокусируйтесь на разработке программного продукта, а не на мелких технических деталях; §С.1.
2. Следование стандарту само по себе не гарантирует переносимости; §С.2.
3. Избегайте неопределенного поведения (включая нестандартные расширения); §С.2.
4. Локализуйте поведение, зависящее от реализации; §С.2.
5. Используйте ключевые слова и диграфы в системах, в которых отсутствуют {, }, [,], | и !, а где отсутствует \ — применяйте триграфы; §С.3.1.
6. Для расширения круга восприятия текста программы используйте лишь символы ANSI; §С.3.3.
7. Числовому представлению символов предпочитайте escape-последовательности; §С.3.2.
8. Не полагайтесь на знаковый или беззнаковый характер типа ***char***; §С.3.4.
9. Если не уверены в типе целого литерала, явно применяйте необходимый суффикс; §С.4.
10. Избегайте неявных преобразований типа, приводящих к потере информации; §С.6.
11. Массивам предпочитайте тип ***vector***; §С.7.
12. Избегайте объединений; §С.8.2.
13. Применяйте битовые поля для представления частей заданной извне раскладки памяти; §С.8.1.
14. Выбирая между разными способами использования памяти, сопоставляйте друг с другом их достоинства и недостатки; §С.9.
15. Не засоряйте глобальное пространство имен; §С.10.1.
16. Если требуется лишь ограниченная область видимости (модуль), а не тип, используйте пространства имен, а не классы; §С.10.3.

17. Не забывайте определять статические члены классовых шаблонов; §C.13.1.
18. Явно маркируйте типы в членах параметров шаблона с помощью ключевого слова ***typename***; §C.13.5.
19. Если возникла необходимость в явной квалификации аргументами шаблона, применяйте ключевое слово ***template*** для явного обозначения шаблонных членов класса; §C.13.6.
20. Определяйте шаблоны, минимизируя их зависимость от контекста конкретизации; §C.13.8.
21. Если конкретизация шаблона занимает слишком много времени, попробуйте конкретизировать их явным образом; §C.13.10.
22. Если нужна предсказуемая последовательность компиляции частей программы, попробуйте применить явную конкретизацию; §C.13.10.

Приложение D

Локализация

Находясь в Риме — поступай как Римлянин.
— Поговорка

Национальные особенности — класс ***locale*** — именованные локализации — создание локализаций — копирование и сравнение локализаций — контексты локализации ***global***() и ***classic***() — класс ***facet*** — доступ к фасетам локализаций — простой пользовательский фасет — стандартные фасеты — сравнение строк — ввод/вывод чисел — ввод/вывод денежных величин — ввод/вывод дат и времени — низкоуровневые операции со временем — класс ***Date*** — классификация символов — преобразование кодов символов — сообщения — советы — упражнения.

D.1. Национальные особенности

Объекты типа ***locale*** (*объекты локального контекста*) призваны учитывать местные и национальные особенности программ, такие как сравнение и сортировка строк, формат вывода чисел и представление символов на внешних носителях. Для более глубокой локализации программ можно расширять ***locale*** новыми *фасетами* (***facet***)[1], отражающими дополнительные национальные особенности, которые исходно не включены в стандартную библиотеку, например, почтовые индексы или телефонные номера. Объекты локализации в стандартной библиотеке в основном применяются для управления отображением выводимых в поток ***ostream*** данных, а также для контроля формата данных, вводимых в рамках потока ***istream***.

В разделе §21.7 рассказано, как изменить локализацию для потока; в данном приложении рассматривается, как объекты локализации конструируются из фасетов, а также объясняются механизмы влияния локализации на поток. Также здесь рассматривается определение фасетов, перечисляются стандартные фасеты, задающие специфические свойства потоков, изложены технологии реализации и приме-

[1] Фасет — грань, аспект чего-либо. — *Прим. ред.*

нения объектов ***locale*** и ***facet***. Стандартные библиотечные средства для представления даты и времени рассматриваются в контексте ввода/вывода.

Весь этот материал организован следующим образом:

- §D.1. Описывает основные идеи, касающиеся локализации программ под национальные особенности с помощью объектов ***locale***.
- §D.2. Представляет класс ***locale***.
- §D.3. Представляет класс ***facet***.
- §D.4. Содержит обзор стандартных фасетов с описанием каждого из них:
  - §D.4.1. Сравнение строк.
  - §D.4.2. Ввод и вывод числовых значений.
  - §D.4.3. Ввод и вывод финансовой информации.
  - §D.4.4. Ввод и вывод дат и времени.
  - §D.4.5. Классификация символов.
  - §D.4.6. Преобразование кодов символов.
  - §D.4.7. Сообщения.

Общее понятие локализации программ не относится непосредственно к языку C++. Операционные системы и среды разработки также оперируют понятием локализации. В принципе, локализация системы относится ко всем программам независимо от того, на каком языке они написаны. Таким образом, понятие контекста локализации стандартной библиотеки C++ можно рассматривать как стандартный и переносимый способ доступа к информации, имеющей существенно различное представление в разных системах.

D.1.1. Программирование национальных особенностей

Рассмотрим вопрос о написании программ, которые могут использоваться в разных странах. Особенности написания таких программ часто называют *интернационализацией* (*internationalization*) — когда внимание фокусируется на попытках обеспечить применение программы в нескольких странах), или *локализацией* (*localization*) — когда внимание фокусируется на попытке обеспечить наилучшую адаптацию программы к конкретным локальным (национальным) особенностям. Многие объекты, которыми программа манипулирует, в разных странах принято выводить по-разному. Мы можем справиться с этой проблемой, учтя эти особенности в наших процедурах ввода/вывода:

```
void print_date(const Date& d)      // print in the appropriate format
{
  switch(where_am_I)                // пользовательский индикатор стиля
  {
   case DK:                         // например, 7. marts 1999
     cout << d.day() << ". " << dk_month[d.month()] << " " << d.year();
     break;
   case UK:                         // например, 7 / 3 / 1999
     cout << d.day() << " / " << d.month() << " / " << d.year();
     break;
   case US:                         // например, 3/7/1999
```

```
        cout << d.month() << "/" << d.day() << "/" << d.year();
        break;
        // ...
    }
}
```

Этот подход решает задачу. Однако он достаточно неуклюж, к тому же его нужно последовательно применять ко всему выводу, чтобы полностью выполнить адаптацию под локальные условия. Но хуже всего то, что если нам понадобится новый способ вывода даты, придется модифицировать исходный код. Казалось бы, с проблемой можно справиться, создав иерархию классов (§12.2.4). Однако информация в ***Date*** не зависит от способа, котором мы хотим ее выводить. Следовательно, нам не подходит иерархия типов ***Date***, например, ***US_date***, ***UK_date*** и ***JP_date***. Скорее, нам нужны разные способы вывода дат: например, вывод в американском стиле, вывод в британском стиле или вывод в японском стиле; см. §D.4.4.5.

Иные проблемы возникают в рамках подхода «пусть пользователь сам пишет функции ввода/вывода, которые учитывают необходимые национальные особенности»:

1. Без помощи стандартной библиотеки прикладной программист вряд ли легко сможет изменить представление встроенных типов эффективным и переносимым образом.
2. Нахождение каждой операции ввода/вывода (и каждой операции подготовки ввода/вывода с учетом национальных особенностей) в большой программе не всегда достижимо.
3. Иногда у нас нет возможности переписать программу для учета нового соглашения — а если и есть, то вряд ли это для нас будет приятным занятием.
4. Заставлять всех пользователей реализовывать учет разных национальных особенностей весьма накладно.
5. Разные программисты будут по-разному реализовывать низкоуровневые средства учета национальных особенностей, так что программы, работающие с одинаковой информацией будут различаться по случайным причинам. Из-за этого программистам, поддерживающим код, полученный из разных источников, придется знакомиться с самыми разными программными решениями, что не только утомительно, но и чревато ошибками.

Как следствие, стандартная библиотека обеспечивает расширяемый способ учета национальных соглашений. На эту схему работы опирается библиотека ввода/вывода (§21.7) как для встроенных типов, так и для типов, определяемых пользователем. Рассмотрим для примера простой цикл копирования пар (***Date, double***), которые могут представлять совокупность измерений или транзакций:

```
void cpy(istream& is, ostream& os)    // copy (Date,double) stream
{
    Date d;
    double volume;

    while(is>>d>>volume) os << d << ' '<< volume << '\n';
}
```

Само собой разумеется. что настоящая программа будет что-нибудь делать с записями, а также будет более внимательна к обработке ошибочных ситуаций.

Как заставить эту программу прочитать файл с французскими соглашениями (дробная часть отделяется с помощью запятой; например, 12,5 означает «двенадцать с половиной») и записать его в американском формате? Для этого мы можем предоставить объекты локального контекста и операции ввода/вывода так, что функция *cpy()* будет выполнять преобразование между указанными соглашениями:

```
void f(istream& fin, ostream& fout, istream& fin2, ostream& fout2)
{
  fin.imbue(locale("en_US"));     // американский английский
  fout.imbue(locale("fr"));       // французский
  cpy(fin,fout);                  // читаем американский английский, пишем французский

  fin2.imbue(locale("fr"));       // французский
  fout2.imbue(locale("en_US"));   // американский английский
  cpy(fin2,fout2);                // читаем французский, пишем американский английский
}
```

Тогда для потоков

```
Apr 12,    1999    1000.3
Apr 13,    1999    345.45
Apr 14,    1999    9688.321
...
3juillet   1950    10,3
3 juillet  1951    134,45
3 juillet  1952    67,9
```

эта программа породит вывод

```
12 avril   1999    1000,3
13 avril   1999    345,45
14 avril   1999    9688,321
...
July 3     1950    10.3
July 3,    1951    134.45
July 3,    1952    67.9
```

Оставшаяся часть настоящего приложения в основном описывает механизмы, делающие подобную схему работы возможной, и объясняет, как их нужно применять. Отметим, что у большинства программистов нет причин разбираться в мельчайших деталях локальных контекстов. Многие программисты никогда не манипулируют объектами *locale* явным образом, а те, что все-таки выполняют такую работу, просто извлекают стандартный *locale* и закрепляют его за потоком (§21.7). В то же время, механизмы, обеспечивающие создание таких объектов и возможность их простого использования представляют из себя небольшой язык программирования.

А если программа или система оказались успешными, они в обязательном порядке будут использоваться людьми, потребности и предпочтения которых проектировщики не предвидели. Большинство успешных программ будут использоваться в странах, естественные языки и наборы символов в которых отличаются от таковых для оригинальных (исходных) проектировщиков и программистов. Широкое

применение программы является признаком успеха программы, так что проектирование и программирование, нацеленные на преодоление языковых и культурных барьеров, являются необходимыми предпосылками для успешной работы.

Концепция локализации (интернационализации) проста. Но многочисленные практические детали изрядно запутывают дизайн и реализацию объектов ***locale***:

1. Объекты ***locale*** должны инкапсулировать национальные стандарты, такие как, например, форма отображения дат. Подобные национальные соглашения варьируются непостижимым и нерегулярным образом. Они не имеют ничего общего с языками программирования, и поэтому язык программирования не может стандартизовать их.
2. Концепция локализации должна быть расширяемой, поскольку невозможно исчерпывающе перечислить все особенности, присущие всем национальным культурам, и представляющие интерес для пользователей C++.
3. Объекты ***locale*** используются в операциях ввода/вывода, от которых требуется высокая эффективность на этапе выполнения.
4. Локализация должна быть прозрачна для большинства программистов, желающих просто воспользоваться правильно работающим вводом/выводом, и не желающих разбираться в том, как все это в точности устроено.
5. Локализация должна быть доступна разработчикам средств, имеющих дело с зависящей от национальных особенностей информацией в рамках, выходящих за пределы стандартных потоков ввода/вывода.

Проектирование программы, имеющей дело с вводом/выводом, имеет выбор между управлением форматированием при помощи «обычного кода», или при помощи объектов ***locale***. Первый (более традиционный) подход приемлем в случаях, когда мы уверены, что каждая операция ввода может быть легко преобразована из одной системы соглашений в другую. Однако если нужно менять формы отображения встроенных типов, или если требуются разные символьные наборы, или набор соглашений может расширяться, то становится привлекательным подход с объектами ***locale***.

В общем случае ***locale*** составляется из фасетов, контролирующих индивидуальные аспекты соглашений, таких как символы-разделители для отображения чисел с плавающей запятой (***decimal_point***(); §D.4.2) или формат, применяемый для чтения денежных величин (***moneypunct***; §D.4.3). Фасет является объектом класса, производного от класса ***locale::facet*** (§D.3). Мы можем рассматривать ***locale*** как контейнер фасетов (§D.2, §D.3.1).

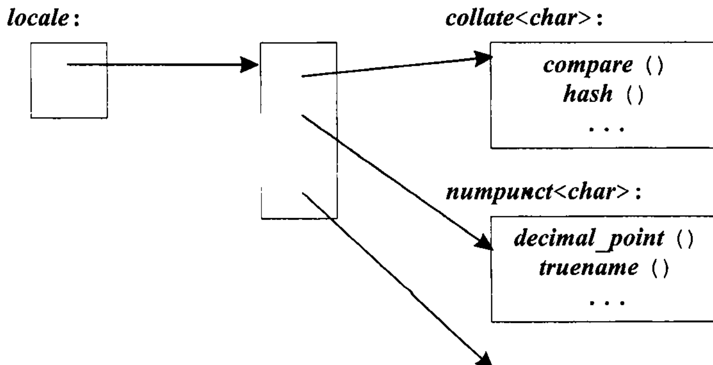
D.2. Класс locale

Класс ***locale*** и связанные с ним средства представлены в заголовочном файле <***locale***>:

```
class std::locale
{
public:
  class facet;               // тип для представления фасетов; §D.3
  class id;                  // тип для идентификации локализаций; §D.3
  typedef int category;      // тип для разбиения фасетов на категории
```

```
static const category    // настоящие значения зависят от реализации
        none = 0,
        collate = 1,
        ctype = 1<<1,
        monetary = 1<<2,
        numeric = 1<<3,
        time = 1<<4,
        messages = 1<<5,
        all = collate | ctype | monetary | numeric | time | messages;

locale () throw () ;                     // копирует глобальную локализацию (§D.2.1)
locale (const locale& x) throw () ;      // копирует x
explicit locale (const char* p) ;        // копия локализации с именем p (§D.2.1)

~locale () throw () ;

locale (const locale& x, const char* p, category c) ;       // копия x плюс фасеты c из p
locale (const locale& x, const locale& y, category c) ;     // копия x плюс фасеты c из y

template<class Facet> locale (const locale& x, Facet* f) ; // копия x плюс фасет f
template<class Facet> locale combine (const locale& x) ;   // копия *this плюс Facet из x

const locale& operator= (const locale& x) throw () ;

bool operator== (const locale&) const;      // сравнение локализаций
bool operator!= (const locale&) const;

string name () const;                       // имя данной локализации (§D.2.1)

template<class Ch, class Tr, class A>       // сравнение строк в данной локализации
bool operator () (const basic_string<Ch,Tr,A>&, const basic_string<Ch,Tr,A>&) const;

static locale global (const locale&) ;      // уст-ка новой глоб-ой локал-ии и возвр. старой
static const locale& classic () ;           // "классическая" локализация в C-стиле

private:
  // representation
};
```

Тип locale можно рассматривать как интерфейс к ***map<id,facet*>***, то есть как нечто, позволяющее нам использовать ***locale::id*** для нахождения соответствующего объекта класса, производного от ***locale::facet***. Истинная реализация класса ***locale*** является эффективным вариантом этой идеи. Схема выглядит примерно следующим образом:



Здесь ***collate<char>*** и ***numpunct<char>*** являются стандартными библиотечными фасетами (§D.4). Как и любые фасеты, они наследуют от ***locale::facet***.

Полагается, что объекты ***locale*** (объекты локализации, объекты локального контекста) можно легко и дешево копировать. Отсюда следует, что класс ***locale*** почти наверняка является дескриптором к специализированному ***map<id,facet*>***, представляющему собой основную часть реализации. Доступ к фасетам объектов локализации должен осуществляться быстро и эффективно, так что специализированный ***map<id,facet*>*** должен быть оптимизирован для быстрого доступа наподобие массива. Доступ к фасету объекта ***locale*** выполняется в нотации ***use_facet<Facet>(loc)*** (см. §D.3.1).

Стандартная библиотека предоставляет богатый набор фасетов. Чтобы помочь программисту ориентироваться в стандартных фасетах, они группируются в категории, такие как ***numeric*** или ***collate*** (§D.4).

Программист может заменять фасеты в существующих категориях (§D.4, §D.4.2.1). Нет, однако, возможности добавлять новые категории (нет возможности определить новую категорию программным путем). Понятие категории фасетов относится только к фасетам стандартной библиотеки и не является расширяемым. Фасет не обязан принадлежать к какой-либо категории, так что многие пользовательские фасеты и не принадлежат им.

По большей части локализация (неявно) используется в потоках ввода/вывода. У каждого ***istream*** и ***ostream*** есть свой ***locale***. По умолчанию, поток получает глобальный ***locale*** (§D.2.1) в момент создания потока. Задать локализацию потока можно при помощи функции ***imbue()***, а извлекается копия ***locale*** потока при помощи функции ***getloc()*** (§21.6.3).

D.2.1. Локальные контексты с заданными именами

Объекты ***locale*** конструируются из других ***locale*** и фасетов. Самым простым способом их создания является копирование. Например:

```
locale loc0;                         // копия текущей глобальной локализации (§D.2.3)

locale loc1 = locale();              // копия текущей глобальной локализации (§D.2.3)
locale loc2("");                     // копия "локализации, предпочитаемой пользователем"

locale loc3("C");                    // копия "C"-локализации
locale loc4 = locale::classic();     // копия "C"-локализации

locale loc5("POSIX");                // копия "POSIX"-локализации, задаваемой реализацией
```

Смысл ***locale("C")*** задается стандартом как классический контекст локализации языка С; именно он подразумевался на протяжении всей книги. Остальные имена контекстов локализации определяются реализациями.

Полагается, что ***locale("")*** задает контекст локализации, предпочтительный для пользователя системы, который формируется не языковыми, а системными средствами.

В большинстве операционных систем имеются свои собственные средства формирования контекста локализации программы. Часто локализация фиксируется в момент начала работы с системой. Например, можно предположить, что пользователь, выбравший аргентинский испанский в качестве рабочего языка по умолчанию, получит для ***locale("")*** именно ***locale("es_AR")***. Беглый взгляд на одну из

моих систем обнаружил 51 мнемоническое имя для контекстов локализации, таких как ***POSIX***, ***de***, ***en_UK***, ***en_US***, ***es***, ***es_AR***, ***fr***, ***sv***, ***da***, ***pl*** и ***iso_8859_1***. Стандарт ***POSIX*** рекомендует начинать их именем языка (буквами нижнего регистра), за которыми может следовать название страны прописными буквами, после чего может следовать спецификатор кодировки, например, ***jp_JP.jit***. Однако этот стандарт не распространяется на все платформы, на некоторых из которых, среди прочего, обнаружил имена ***g***, ***uk***, ***us***, ***s***, ***fr***, ***sw*** и ***da***. Стандарт языка C++ не определяет смысла объектов локализации для заданных стран или конкретных естественных языков, что, однако, может стандартизоваться в рамках тех или иных платформ. Поэтому при обращении к именованным контекстам локализации программисту нужно ознакомиться с соответствующей документацией по его системе (и, возможно, поэкспериментировать).

Как правило, нежелательно непосредственно в тексте программы располагать строки с именами локализаций. Это ограничивает переносимость программы и заставляет программиста просматривать весь ее текст при необходимости адаптации программы к новой среде. Вместо этого лучше динамически считать системную информацию о локализации с помощью ***locale***(""), или предоставить опытному пользователю возможность выбора альтернативной локализации путем ввода соответствующей строки. Например:

```
void user_set_locale(const string& question_string)
{
  cout << question_string;              // "Нужна другая локализация? - введите ее имя"
  string s;
  cin >> s;
  locale::global(locale(s.c_str()));    // установить указанную пользователем
                                        // локализацию в качестве глобальной
}
```

Менее подготовленному пользователю желательно предоставить выбор возможных альтернатив из заранее сформированного списка строк с допустимыми именами. Программа должна знать, где система хранит сведения о локализации.

Если строковый аргумент ссылается на недействительную локализацию, конструктор генерирует исключение ***runtime_error*** (§14.10). Например:

```
void set_loc(locale& loc, const char* name)
try
{
  loc = locale(name);
}
catch(runtime_error)
{
  cerr << "locale \"" << name << "\" isn't defined\n";
  // ...
}
```

Если контекст локализации именованный, то функция ***name***() вернет его имя, а если неименованный — то ***string***("*"). Строку с именем можно использовать для отладки. Например:

```
void print_locale_names(const locale& my_loc)
{
  cout<< "name of current global locale: " << locale().name() << "\n";
  cout<< "name of classic C locale: " << locale::classic().name() << "\n";
  cout<< "name of ''user's preferred locale'': " <<locale("").name()<< "\n";
  cout<< "name of my locale: " << my_loc.name() << "\n";
}
```

Контексты локализации с идентичными именами (отличными от значения по умолчанию *string*("*")) считаются совпадающими. В то же время, операции == или != предоставляют более непосредственный способ их сравнения.

Вызов *locale*(*loc*, "*Foo*", *cat*) создает контекст, подобный *loc*, но фасеты берутся из категории *cat* контекста *locale*("*Foo*"). Результирующий контекст имеет строку имени лишь в том случае, если она есть у исходного контекста *loc*. Стандарт не определяет, какова эта строка на самом деле, но предполагается, что она будет отличаться от таковой у контекста *loc*. Одно из возможных решений — формирование строки из исходной строки контекста *loc* плюс имя "*Foo*". Например, если у *loc* строка имени есть "*en_UK*", то новый контекст получит строку "*en_UK*:*Foo*".

Обобщим сведения о строке имени вновь создаваемого контекста следующим образом:

Локализации	Строка имени
locale("*Foo*")	"*Foo*"
locale(*loc*)	*loc*.*name*()
locale(*loc*, "*Foo*", *cat*)	Новая строка имени, если у *loc* есть строка имени; иначе – *string*("*")
locale(*loc*, *loc2*, *cat*)	Новая строка имени, если у *loc* и *loc2* есть строка имени; иначе – *string*("*")
locale(*loc*, *Facet*)	*string*("*")
loc.*combine*(*loc2*)	*string*("*")

Не существует программных средств, позволяющих для вновь созданного контекста задать строку имени локализации в С-стиле. Строки имен либо определяются в рамках системы, либо создаются в виде комбинаций таких имен конструкторами объектов *locale*.

D.2.1.1. Конструирование новых объектов локализации

Новый контекст локализации создается на основе существующего посредством добавления или замены фасетов. В типичном случае это приводит к незначительной модификации исходного контекста. Например:

```
void f(const locale& loc, const My_money_io* mio) // My_money_io определен в §D.4.3.1
{
  locale loc1(locale("POSIX"),loc,locale::monetary);    // monetary фасеты из loc
  locale loc2 = locale(locale::classic(),mio);          // classic плюс mio
  // ...
}
```

Здесь ***loc1*** является модифицированной копией контекста ***POSIX***, в которую добавляются «денежные» фасеты из ***loc*** (§D.4.3). Аналогично, ***loc2*** является копией контекста в С-стиле, модифицированной для использования ***My_money_io*** (§D.4.3.1). Если аргумент типа ***Facet**** (в нашем случае это ***My_money_io***) равен ***0***, то результирующий контекст будет просто копией аргумента типа ***locale***.

В записи

```
locale (const locale& x, Facet* f) ;
```

аргумент ***f*** должен идентифицировать конкретный тип фасета. Просто ***facet**** не проходит. Например:

```
void g (const locale::facet* mio1, const My_money_io* mio2)
{
  locale loc3 = locale (locale::classic () , mio1) ;    // error: не известен тип фасета
  locale loc4 = locale (locale::classic () , mio2) ;    // ok: тип фасета известен
  // ...
}
```

Причина в том, что ***locale*** использует аргумент ***Facet**** для определения типа фасета на этапе компиляции. А именно, реализация ***locale*** использует тип, идентифицирующий фасет, ***facet::id*** (§D.3), для поиска фасета в контексте локализации.

Обратите внимание, что конструктор

```
template<class Facet> locale (const locale& x, Facet* f) ;
```

является единственным языковым механизмом, позволяющим программисту задать фасет, который будет использоваться в объекте локализации. Именованные контексты локализации предоставляются непосредственно реализациями (§D.2.1). Эти именованные локализации можно извлекать из системной среды. Если программист сможет изучить и понять эти механизмы, он сможет таким способом добавлять новые контексты локализации (§D.6[11,12]).

Набор конструкторов для ***locale*** спроектирован таким образом, чтобы тип каждого фасета становился известен либо путем выведения типа (из параметра ***Facet*** шаблона), либо благодаря тому, что он достается от другого объекта локализации (и который знает этот тип). Задание аргумента ***category*** косвенно определяет тип фасета, потому что в контексте локализации типы фасетов в категориях известны. Это подразумевает, что в классе ***locale*** типы фасетов известны, что позволяет манипулировать ими с минимальными накладными расходами.

Для идентификации типов фасетов класс ***locale*** использует поле ***locale::id*** (§D.3).

Бывает полезно создать объект локализации, во всем являющийся копией другого объекта локализации за исключением того, что некоторые его фасеты взяты из третьего контекста локализации. Это реализуется шаблонной функцией-членом ***combine()***. Например:

```
void f (const locale& loc, const locale& loc2)
{
  locale loc3 = loc.combine<My_money_io> (loc2) ;
  // ...
}
```

Результирующий объект ***loc3*** ведет себя как ***loc***, только он использует копию ***My_money_io*** (§D.4.3.1) из ***loc2*** для форматирования ввода/вывода денежных величин. Если в ***loc2*** нет ***My_money_io***, то функция ***combine***() сгенерирует исключение ***runtime_error*** (§14.10). Результат работы функции ***combine***() не имеет строки имени.

D.2.2. Копирование и сравнение контекстов локализации

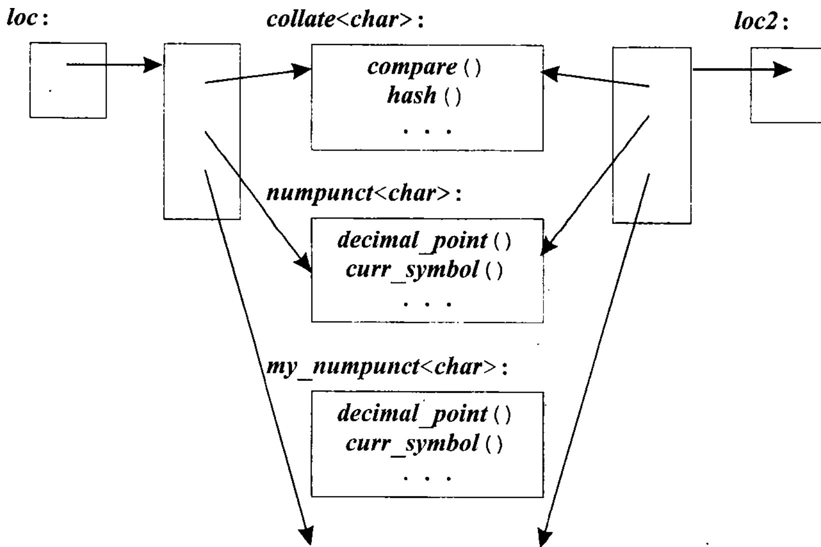
Объекты ***locale*** копируются при инициализации и в операциях присваивания. Например:

```
void swap (locale& x, locale& y)          // как std::swap()
{
  locale temp = x;
  x = y;
  y = temp;
}
```

Копия объекта ***locale*** считается равной оригиналу, но при этом сама она является отдельным независимым объектом. Например:

```
void f(locale* my_locale)
{
  locale loc = locale::classic();          // "C" locale

  if(loc != locale::classic())
  {
    cerr << "implementation error: send bug report to vendor\n";
    exit(1);
  }

  if(&loc != &locale::classic()) cout << "no surprise: addresses differ\n";
  locale loc2 = locale(loc, my_locale, locale::numeric);

  if(loc== loc2)
  {
    cout << "my numeric facets are the same as classic()'s numeric facets\n";
    // ...
  }
  // ...
}
```

Если ***my_locale*** имеет отвечающий за числовую пунктуацию фасет ***my_numpunct<char>***, отличный от стандартного ***numpunct<char>*** из ***classic***(), то результирующие контексты локализации можно отобразить следующим образом:



Не существует способа модификации *locale*. Вместо этого операции класса *locale* позволяют создавать новые объекты локализации на базе уже существующих. Тот факт, что объекты локализации неизменяемы после создания, очень важен для обеспечения эффективности на этапе выполнения. Это позволяет пользователю вызывать виртуальные функции фасетов и кэшировать возвращаемые значения. Например, *istream* может знать, какой символ используется в качестве десятичной запятой, или как представляется значение *true*, не вызывая *decimal_point*() каждый раз при чтении числа и *truename*() при чтении в логическую переменную (§D.4.2). Только вызов *imbue*() для потока может заставить изменить значения, возвращаемые этими вызовами.

D.2.3. Контексты global() и classic()

Понятие текущей локализации программы поддерживается функцией *locale*(), возвращающей копию текущей локализации, а также функцией *locale*::*global*(), устанавливающей контекст локализации в качестве текущего. Текущую локализацию обычно называют глобальной, имея в виду то, что чаще всего она реализуется с помощью глобального (или статического) объекта.

Глобальная локализация используется неявным образом при инициализации потока, то есть к каждому вновь создаваемому потоку прикрепляется (*imbue*(); §21.1, §21.6.3) копия *locale*(). Изначально, глобальная локализация есть стандартная локализация языка С, *locale*::*classic*().

Статическая функция-член *locale*::*global*() позволяет программисту задать контекст локализации, который будет использоваться как глобальный. Эта функция возвращает предыдущий контекст локализации, использовавшийся в качестве глобального контекста, что позволяет при необходимости восстановить его позже. Например:

```
void f(const locale& my_loc)
{
  ifstream fin1(some_name);              // к fin1 прикреплена глобальная локал-ция
```

```
    locale old_global = locale::global(my_loc);   // устанавливаем новую глоб-ную локал-цию
    ifstream fin2(some_other_name);               // к fin2 прикреплена локал-ция my_loc
    // ...
    locale::global(old_global);                   // восстанавливаем стар. глоб-ную локал-цию
}
```

Если в локализации x имеется строка имени, то ***locale*::*global*(*x*)** также устанавливает глобальную локализацию языка C. Это означает, что если программа на C++ вызывает зависимую от контекста локализации стандартную библиотечную функцию языка C, то в такой смешанной среде трактовка локализации будет согласованной.

Однако если локализация x строки имени не содержит, то неизвестно, влияет ли вызов ***locale*::*global*()** на глобальный контекст локализации языка C. Это означает, что в таком случае программа на языке C++ не может надежным и переносимым образом установить для средств языка C локализацию, которая не была получена от системной среды. Также не существует стандартного способа, которым программа на языке C могла бы установить глобальную локализацию C++ (кроме как вызвать для этого функцию C++). В общем, в смешанной среде программ на языках C и C++ использование для языка C глобальной локализации, отличной от ***global*()**, чревато ошибками.

Задание глобальной локализации не влияет на уже существующие потоки ввода/вывода; они по-прежнему используют те локализации, которые были закреплены за ними. Например, на поток ***fin1*** не повлияли последующие манипуляции с глобальной локализацией, в результате которых за потоком ***fin2*** была закреплена локализация ***my_loc***.

Манипуляции с глобальной локализацией страдают общим пороком работы с глобальными объектами — очень трудно (если возможно) проконтролировать, на что может повлиять изменение этих объектов. Поэтому вызовы ***global*()** лучше всего минимизировать и четко локализовать их в нескольких разделах кода, подчиняющихся простой стратегии этих изменений. Возможность прикреплять специфические контексты локализации к отдельным потокам (***imbue*()**; §21.6.3) существенно упрощает дело, но все равно, изобилие разбросанных по программе явных ссылок на контексты локализации сильно затрудняет сопровождение программы.

D.2.4. Сравнение строк

Сравнение двух строк в соответствии с конкретной локализацией является, наверное, самой распространенной задачей, явно использующей контекст локализации. Поэтому данная операция непосредственно предоставляется классом ***locale***, чтобы избавить пользователей от необходимости создавать собственные функции сравнения из фасета ***collate*** (§D.4.1). Чтобы функцию сравнения из класса ***locale*** можно было использовать непосредственно в качестве предиката (§18.4.2), она определена там как ***operator*()()**. Например:

```
void f(vector<string>& v, const locale& my_locale)
{
    sort(v.begin(), v.end());                 // sort с применением < для сравнения элементов
    // ...
    sort(v.begin(), v.end(), my_locale);      // sort согласно правилам my_locale
    // ...
}
```

По умолчанию, стандартная библиотечная функция ***sort*()** использует операцию < с числовыми значениями символьного набора конкретной реализации (§18.7, §18.6.3.1).

Заметьте, что в рамках контекстов локализации сравниваются объекты типа ***basic_string***, а не С-строки.

D.3. Фасеты

Фасеты являются объектами класса, производного от класса ***facet*** (объявленного в классе ***locale***):

```
class std::locale::facet
{
protected:
  explicit facet(size_t r = 0) ;  // "r==0" -локал-я управляет временем жизни этого фасета
  virtual ~facet() ;              // обратите внимание: protected деструктор

private:
  facet(const facet&) ;           // не определен
  void operator= (const facet&) ; // не определен
  // представление класса
}
```

Операции копирования объявлены закрытыми и оставлены неопределенными для предотвращения копирования (§11.2.2).

Класс ***facet*** спроектирован в качестве базового, и в нем нет открытых функций. Его конструктор объявлен как ***protected***, чтобы нельзя было создать объекты типа «просто ***facet***», а деструктор объявлен виртуальным для обеспечения корректного уничтожения объектов производных классов.

Предполагается, что оперировать фасетами будут объекты ***locale*** при помощи указателей. Нулевой аргумент конструктора класса ***facet*** означает, что ***locale*** обязан удалить фасет после исчезновения последней ссылки на него. Напротив, ненулевой аргумент конструктора гарантирует, что ***locale*** никогда не удалит фасет. Это значение аргумента предназначено для редкого случая, когда сам программист управляет временем жизни фасета (а не локализация). Например, мы можем попытаться создать объекты стандартного класса фасетов ***collate_byname<char>*** (§D.4.1.1) следующим образом:

```
void f(const string& s1, const string& s2)
{
  // нормальный случай: (умолчательный) аргумент 0 - locale отвечает за удаление:
  collate<char>* p = new collate_byname<char> ("pl") ;
  locale loc(locale() ,p) ;

  // редкий случай: аргумент 1 - пользователь отвечает за удаление:
  collate<char>* q = new collate_byname<char> ("ge",1) ;

  collate_byname<char> bug1("sw") ;     // error: невозм-но унич-ить лок-ную перем-ую
  collate_byname<char> bug2("no",1) ;   // error: невозм-но унич-ить лок-ную перем-ую
  //...

  // q нельзя удалить: у collate_byname<char> деструктор protected
  // нельзя delete p; locale управляет удалением *p
}
```

Стандартные фасеты полезны, когда управляются контекстами локализации (как базовые классы), и очень редко — в других случаях.

Фасет *_byname*() является фасетом из именованного локального контекста системной среды (§D.2.1).

Чтобы можно было осуществлять поиск фасета в рамках контекста локализации функциями *has_facet*() и *use_facet*(), для каждого типа фасета должен определяться идентификатор *id*:

```
class std::locale::id
{
public:
  id();

private:
  id(const id&);                  // не определен
  void operator=(const id&);      // не определен
  // представление
};
```

Операции копирования объявлены закрытыми и оставлены неопределенными для предотвращения копирования (§11.2.2).

Класс *id* задуман для того, чтобы пользователь мог определить статический член типа *id* в каждом классе, предоставляющем новый интерфейс фасета (например, см. §D.4.1). Внутренние механизмы класса *locale* используют *id* для идентификации фасетов (§D.2, §D.3.1). В очевидной реализации *id* используется как индекс вектора указателей на фасеты, реализуя тем самым эффективный *map<id,facet*>*.

Поля данных, необходимые для формирования (производного) фасета определяются в производных классах, а не в самом базовом классе *facet*. Это означает, что программист, определяющий фасет, имеет полный контроль над данными и что любое количество данных может быть задействовано для реализации концепции фасета.

Заметьте, что все функции-члены определяемого пользователем фасета должны быть константными. В общем случае, фасеты полагаются неизменяемыми (§D.2.2).

D.3.1. Доступ к фасетам класса locale

Доступ к фасетам класса *locale* осуществляется с помощью функционального шаблона *use_facet*. С помощью функционального шаблона *has_facet* мы можем узнать, имеется ли конкретный фасет в заданном контексте локализации:

```
template<class Facet> bool has_facet(const locale&) throw();
template<class Facet> const Facet& use_facet(const locale&); // может генер-ть bad_cast
```

Можно считать, что эти шаблонные функции осуществляют поиск параметра шаблона *Facet* в аргументе *locale*. Можно, однако, взглянуть на функциональный шаблон *use_facet* как на своего рода явное приведение типа от *locale* к конкретному фасету. Эта возможность базируется на том, что у *locale* только один фасет данного типа. Например:

```
void f(const locale& my_locale)
{
  char c = use_facet<numpunct<char> >(my_locale).decimal_point(); // используем
                                                                  // стандартный фасет
   // ...

  If(has_facet<Encrypt>(my_focale))            // в my_locale есть фасет Encrypt?
  {
    const Encrypt& f = use_facet<Encrypt>(my_locale);   // извлекаем фасет Encrypt
    const Crypto c = f.get_crypto();                     // используем фасет Encrypt
    // ...
  }
  // ...
}
```

Обратите внимание на то, что ***use_facet*** возвращает ссылку на константный фасет, так что мы не можем присвоить этот возврат неконстантным переменным. Это разумно, поскольку фасеты полагаются неизменяемыми и содержащими лишь константные методы.

Если мы попытаемся вызвать ***use_facet<X>(loc)***, а ***loc*** не содержит фасет ***X***, то ***use_facet*** сгенерирует исключение ***bad_cast*** (§14.10). Так как гарантируется, что стандартные фасеты доступны во всех контекстах локализации (§D.4), то в этом случае нет необходимости обращаться к ***has_facet***. Для стандартных фасетов ***use_facet*** никогда не сгенерирует исключение ***bad_cast***.

Как могут быть реализованы ***use_facet*** и ***has_facet***? Вспомним, что ***locale*** можно рассматривать как ***map<id,facet*>*** (§D.2). Отталкиваясь от полученного в параметре шаблона ***Facet*** типа фасета, реализации ***use_facet*** или ***has_facet*** могут обратиться к ***Facet::id*** и использовать его для поиска соответствующего фасета. Упрощенные реализации ***has_facet*** и ***use_facet*** могут выглядеть следующим образом:

```
// псевдореализация: пусть locale содержит map<id,facet*> названный facet_map
template<class Facet> bool has_facet(const locale& loc) throw()
{
  const locale::facet* f = loc.facet_map[Facet::id];
  return f ? true : false;
}

template<class Facet> const Facet& use_facet(const locale& loc)
{
  const locale::facet* f = loc.facet_map[Facet::id];
  if(f) return static_cast<const Facet&>(*f);
  throw bad_cast();
}
```

Взглянув по-другому на механизм ***facet::id***, можно реализовать некую форму полиморфизма на этапе компиляции. Очень близкий к ***use_facet*** результат можно получить применением операции dynamic_cast. Однако специализированный шаблон ***use_facet*** может быть реализован гораздо эффективнее общего механизма на базе операции ***dynamic_cast***.

На самом деле ***id*** идентифицирует скорее интерфейс и поведение, нежели класс. То есть если интерфейсы двух классов фасетов в точности совпадают при одинако-

вой семантике (по отношению к контексту локализации), они должны идентифицироваться одним и тем же *id*. Например, *collate<char>* и *collate_byname<char>* для *locale* взаимозаменяемы, и потому оба идентифицируются при помощи одного и того же *collate<char>::id* (§D.4.1).

Если мы определяем фасет с новым интерфейсом — например *Encrypt* в функции *f()* — мы должны определить и соответствующий *id* для его идентификации (см. §D.3.2 и §D.4.1).

D.3.2. Простой пользовательский фасет

Стандартная библиотека предоставляет стандартные фасеты, учитывающие наиболее важные аспекты национальных особенностей, такие как наборы символов и ввод/вывод чисел. Чтобы познакомиться с механизмом фасетов вне их связи с широко применяемыми типами, влекущими за собой всякие сложности, а также вне забот о высокой эффективности, которые их сопровождают, рассмотрим фасет для тривиального пользовательского типа:

```
enum Season {spring, summer, fall, winter};
```

Это простейший пользовательский тип, который я вообще могу себе представить. Показанный ниже стиль ввода/вывода может использоваться (с минимальными изменениями) для большинства простых пользовательских типов:

```
class Season_io: public locale::facet
{
public:
  Season_io (int i = 0) : locale::facet(i) { }
  ~Season_io () { } // обеспечивает возможность уничтожения объектов Season_io (§D.3)

  virtual const string& to_str (Season x) const = 0;    // строковое представление для x

  // place Season corresponding to s in x:
  virtual bool from_str (const string& s, Season& x) const = 0;
  static locale::id id;                 // объект идентификации фасета (§D.2, §D.3, §D.3.1)
};

locale::id Season_io::id;               // определяем идентифицирующий объект
```

Для простоты данный фасет ограничивается представлениями, использующими лишь *char*.

Класс *Season_io* обеспечивает общий абстрактный интерфейс ко всем фасетам иерархии наследования. Чтобы определить вывод объектов типа *Season* для конкретного контекста локализации, мы наследуем класс от *Season_io*, определяя при этом надлежащим образом функции *to_str()* и *from_str()*.

Вывести *Season* просто. Если у потока имеется фасет *Season_io*, то мы можем использовать его для преобразования времени года[1] в строку. В противном случае мы можем вывести целое значение:

```
ostream& operator<< (ostream& s, Season x)
{
  const locale& loc = s.getloc(); // извлекаем stream's locale (§21.7.1)
```

[1] Время года по-английски — season. — *Прим. ред.*

```
    if(has_facet<Season_io>(loc)) return s << use_facet<Season_io>(loc).to_str(x);
    return s << int(x);
}
```

Обратите внимание, что данная операция << реализована посредством вызова этой же операции, но для других типов. Таким образом, мы извлекаем пользу из простоты вызова операций << по сравнению с прямым доступом к буферам потока ***ostream***, из учета этими операциями контекста локализации и из реализованной ими обработки ошибок. Стандартные фасеты напрямую обращаются к буферам потоков (§D.4.2.2, §D.4.2.3) ради максимальной эффективности и гибкости, но для многих простых пользовательских типов нет никакой необходимости опускаться на уровень абстракции ***streambuf***.

Как обычно, ввод немного сложнее вывода:

```
istream& operator>>(istream& s, Season& x)
{
    const locale& loc = s.getloc();          // извлекаем stream's locale (§21.7.1)
    if(has_facet<Season_io>(loc))             // читаем алфавитное представление
    {
        const Season_io& f = use_facet<Season_io>(loc);
        string buf;
        if(!(s>>buf && f.from_str(buf,x))) s.setstate(ios_base::failbit);
        return s;
    }

    int i;                                   // читаем числовое представление
    s >> i;
    x = Season(i);
    return s;
}
```

Обработка ошибок проста и следует стилю обработки ошибок для встроенных типов. То есть если вводимая строка не соответствует стилю ***Season*** в рамках текущего контекста локализации, поток устанавливается в состояние ***fail***. Если допускаются исключения, то следует обрабатывать исключения ***ios_base::failure*** (§21.3.6).

Вот тривиальная тестовая программа:

```
int main()
{
    Season x;

    // Используем локал-цию по умолчанию (нет фасета Season_io) - ввод/вывод целых:
    cin >> x;
    cout << x << endl;

    locale loc(locale(),new US_season_io);
    cout.imbue(loc);         // используем локализацию с фасетом Season_io
    cin.imbue(loc);          // используем локализацию с фасетом Season_io

    cin >> x;
    cout << x << endl;
}
```

Для входа

```
2
summer
```

эта программа отреагирует следующим выводом:

```
2
summer
```

Для достижения этого результата мы должны определить ***US_season_io*** с целью задания строкового представления времен года и замещения функций интерфейса ***Season_io***, которые преобразуют строковые представления в перечисления и обратно:

```
class US_season_io : public Season_io
{
  static const string seasons [ ] ;

public:
  const string& to_str (Season) const;
  bool from_str (const string&, Season&) const;
  // обратите внимание: отсутствует US_season_io::id
};

const string US_season_io : : seasons [ ] = { "spring", "summer", "fall", "winter" } ;

const string& US_season_io : : to_str (Season x) const
{
  if(x<spring || winter<x)
  {
    static const string ss = "no-such-season";
    return ss;
  }
  return seasons [x] ;
}

bool US_season_io : :from_str (const string& s, Season& x) const
{
  const string* beg = &seasons [spring] ;
  const string* end = &seasons [winter] +1;
  const string* p = find (beg, end, s) ;         // §3.8.1, §18.5.2

  if (p==end) return false;
  x = Season (p-beg) ;
  return true;
}
```

Заметьте, что поскольку ***US_season_io*** является просто реализацией интерфейса ***Season_io***, я не определял id для ***US_season_io***. Вообще, если мы хотим использовать ***US_season_io*** в качестве ***Season_io***, мы не можем задавать для ***US_season_io*** его собственный ***id***. Операции класса ***locale***, например ***has_facet*** (§D.3.1), полагаются на то, что фасеты, реализующие одну и ту же концепцию, идентифицируются одним и тем же ***id*** (§D.3).

Единственным вопросом в данной реализации остается вопрос, что делать, когда для вывода предоставляется некорректный ***Season***? Случаи появления некор-

ректных значений для простых пользовательских типов не столь уж необычны, так что стоит заранее принять во внимание подобную ситуацию. Я мог бы сгенерировать исключение. Но когда имеешь дело с простым выводом, предназначенным для человеческих глаз, полезно подготовить специальное визуальное представление для некорректных значений. Обратите внимание, что при вводе стратегия обработки ошибок передана операции <<, в то время как при выводе эта стратегия оставляется функции фасета *to_str*(). Это было сделано с целью проиллюстрировать возможные варианты проектирования. В настоящих, промышленных проектах функции фасета либо реализуют обработку ошибок как ввода, так и вывода, либо перепоручают все это операциям >> и <<.

Данный проект для *Season_io* полагается на то, что производные классы предоставляют специфичные для локализации строки. В альтернативном дизайне *Season_io* сам мог бы извлекать строки из соответствующего хранилища (см. §D.4.7). Создание класса *Season_io*, которому строки времен года передаются в качестве аргументов конструктора, оставляется в качестве упражнения (§D.6[2]).

D.3.3. Использование локализаций и фасетов

В рамках стандартной библиотеки локализации в первую очередь принимаются во внимание в операциях ввода/вывода. Тем не менее, механизм локализации является общим и расширяемым средством представления информации, чувствительной к локальным и национальным особенностям. Класс *messages* (§D.4.7) является примером фасета, не имеющего никакого отношения ко вводу/выводу. Расширения библиотеки потокового ввода/вывода и средства ввода/вывода, не опирающиеся на потоки, также могут использовать объекты *locale*. Пользователь может использовать контексты локализации для организации произвольной информации, зависимой от национальных и культурных особенностей.

Благодаря общности механизмов локализаций и фасетов возможности определяемых пользователем фасетов не ограничены. Вероятными кандидатами на представление фасетами являются даты, часовые пояса, телефонные номера, номера социального страхования, товарные коды, температурные величины, произвольные пары измеряемых величин (единица измерения, значение), почтовые коды (zip-коды), размеры одежды и т.д.

Как и другие мощные механизмы, фасеты следует применять с определенной осторожностью. Из одного лишь факта, что нечто может быть представлено в виде фасета, не следует, что такое представление будет наилучшим. Для выбора ключевыми, как и всегда, являются следующие факторы: как различные решения влияют на сложность написания кода, легко или сложно читать этот код, насколько сложно поддерживать код в дальнейшем в процессе его эксплуатации и неизбежных модификаций, какова эффективность операций ввода/вывода в плане быстродействия и используемой памяти.

D.4. Стандартные фасеты

Стандартная библиотека предоставляет в заголовочном файле <*locale*> следующие фасеты для локализации *classic*():

Стандартные фасеты (в локализации classic())			
	Категория	**Назначение**	**Фасеты**
§D.4.1	***collate***	сравнение строк	***collate<Ch>***
§D.4.2	***numeric***	числовой ввод/вывод	***numpunct<Ch>***
			num_get<Ch>
			num_put<Ch>
§D.4.3	***monetary***	ввод/вывод денежных сумм	***moneypunct<Ch>***
			moneypunct<Ch,true>
			money_get<Ch>
			money_put<Ch>
§D.4.4	***time***	ввод/вывод времени	***time_get<Ch>***
			time_put<Ch>
§D.4.5	***ctype***	классификация символов	***ctype<Ch>***
			codecvt<Ch,char,mbstate_t>
§D.4.7	***messages***	извлечение сообщений	***messages<Ch>***

В этой таблице ***Ch*** означает ***char*** или ***wchar_t***. Пользователь, которому требуется, чтобы стандартный ввод/вывод работал с другим символьным типом ***X***, должен предоставить подходящие варианты фасетов для ***X***. Например, может потребоваться ***codecvt<X,char,mbstate_t>*** (§D.4.6) для управления преобразованиями между ***X*** и ***char***. Тип ***mbstate_t*** используется для представления состояния сдвига в многобайтных кодировках символов (§D.4.6); ***mbstate_t*** определен в <***cwchar***> и <***wchar.h***>. Эквивалентом ***mbstate_t*** для произвольного символьного типа ***X*** является ***char_traits<X>::state_type***.

В заголовочном файле <***locale***> стандартная библиотека дополнительно представляет следующие фасеты:

Стандартные фасеты			
	Категория	**Назначение**	**Фасеты**
§D.4.1	***collate***	сравнение строк	***collate_byname<Ch>***
§D.4.2	***numeric***	ввод/вывод чисел	***numpunct_byname <Ch>***
			num_get<C,In>
			num_put<C,Out>
§D.4.3	***monetary***	ввод/вывод денег	***moneypunct_byname<Ch, International>***
			money_get<C,In>
			money_put<C,Out>
§D.4.4	***time***	ввод/вывод времени	***time_put_byname <Ch,Out>***
§D.4.5	***ctype***	классификация символов	***ctype_byname <Ch>***
§D.4.7	***messages***	извлечение сообщений	***messages_byname <Ch>***

При конкретизации фасетов из данной таблицы ***Ch*** может быть ***char*** или ***wchar_t***; ***C*** может быть любым символьным типом (§20.1). Параметр ***International*** может иметь значение ***true*** или ***false***; ***true*** означает, что используется международное четырехсимвольное представление валютного знака (§D.4.3.1). Тип ***mbstate_t*** определен в <***cwchar***> и <***wchar.h***>.

Параметры ***In*** и ***Out*** — это итераторы ввода и вывода, соответственно (§19.1, §19.2.1). Придание фасетам с суффиксами ***_put*** и ***_get*** этих параметров позволяет программисту реализовывать фасеты с прямым доступом к нестандартным буферам (§D.4.2.2). Буфера, ассоциированные с ***iostream***, являются буферами потоков, так что тип связанных с ними итераторов есть ***ostreambuf_iterator*** (§19.2.6.1, §D.4.2.2). Как следствие, для обработки ошибок доступна функция ***failed***() (§19.2.6.1).

Фасет ***F_byname*** является производным от фасета ***F***. Фасет ***F_byname*** предоставляет такой же интерфейс, что и ***F***, за исключением того, что добавлен конструктор со строковым аргументом для задания имени локализации (см. §D.4.1). Фасет ***F_byname***(***name***) предоставляет соответствующую семантику для ***F***, определенную в ***locale***(***name***). Идея заключается в том, чтобы извлечь версию стандартного фасета из именованной локализации (§D.2.1) системной среды выполнения программы. Например:

```
void f(vector<string>& v, const locale& loc)
{
    locale d1(loc,new collate_byname<char>("da"));     // сравнение датских строк
    locale dk(d1,new ctype_byname<char>("da"));        // классиф-я датских символов
    sort(v.begin(),v.end(),dk);
    // ...
}
```

Новый контекст локализации ***dk*** будет пользоваться датскими текстовыми строками, но прежними умолчательными соглашениями по числам. Отметим, что поскольку умолчательное значение второго аргумента фасета есть ***0***, то ***locale*** будет управлять временем жизни фасета, созданного операцией ***new*** (§D.3).

Подобно конструкторам класса ***locale*** со строковыми аргументами, конструкторы фасетов с суффиксом ***_byname*** осуществляют доступ к системной среде выполнения программы. Это означает, что они существенно медленнее конструкторов, которым не нужен такой доступ. Почти всегда быстрее сначала создать объект локализации, и лишь после этого выполнять доступ к его фасетам, чем пользоваться фасетами с суффиксом ***_byname*** во многих местах программы. Таким образом, хорошей идеей является однократное чтение фасета из системной среды с последующим многократным использованием его копии во многих местах программы. Например:

```
locale dk("da");    // читаем датскую локал-ю (включая все фасеты) единожды
                    // а затем используем dk и его фасеты по необходимости

void f(vector<string>& v, const locale& loc)
{
    const collate<char>& col = use_facet<collate<char> >(dk);
    const collate<char>& ctyp = use_facet<ctype<char> >(dk);

    locale d1(loc,col);  // сравнение датских строк
    locale d2(d1,ctyp);  // сравнение датских строк и классификация датских символов
```

```
  sort(v.begin(),v.end(),d2);
  // ...
}
```

Понятие категории предоставляет более простой способ манипулирования стандартными фасетами в контекстах локализации. Например, располагая локализацией ***dk***, мы можем создать контекст локализации, в рамках которого строки читаются и сравниваются в соответствии с правилами датского языка (в котором есть три лишних гласных по сравнению с английским языком), а числа сохраняют синтаксис языка C++:

```
locale dk_us(locale::classic(),dk,collate|ctype); // датские буквы, обычные числа
```

Ниже при описании конкретных стандартных фасетов даются дополнительные примеры их использования. В частности, обсуждение ***collate*** (§D.4.1) проявляет многие структурные (архитектурные) аспекты фасетов.

Обратите внимание на то, что стандартные фасеты часто зависят друг от друга. Например, ***num_put*** зависит от ***numpunct***. Чтобы успешно комбинировать и сопоставлять фасеты, а также создавать новые стандартные фасеты, вы должны хорошо знать их индивидуальные свойства. Другими словами, за исключением простых операций, рассмотренных в §21.7, механизмы контекстов локализации не предназначены для их непосредственного использования новичками.

Разработка конкретного фасета весьма трудоемка. Отчасти причина заключается в том, что конкретные национальные и культурные соглашения сами по себе весьма запутанные, а кроме того важно, чтобы средства стандартной библиотеки C++ были совместимы со средствами стандартной библиотеки языка C и со многими другими системными соглашениями и стандартами. Например, разработчику библиотеки было бы неразумно игнорировать средства локализации POSIX.

С другой стороны, среда, формируемая с помощью ***locale*** и фасетов, весьма гибкая и общая. Можно сконструировать фасет для любых данных, а операции фасета позволят выполнять с этими данными любые специфические для них действия. Если фасет не будет слишком стеснен существующими соглашениями, то его дизайн и реализация могут быть простыми и ясными (§D.3.2).

D.4.1. Сравнение строк

Стандартный фасет collate обеспечивает сравнение массивов символов типа ***Ch***:

```
template<class Ch>
class std::collate: public locale::facet
{
public:
  typedef Ch char_type;
  typedef basic_string<Ch> string_type;

  explicit collate(size_t r = 0);

  int compare(const Ch* b, const Ch* e, const Ch* b2, const Ch* e2) const
  {return do_compare(b,e,b2,e2);}

  long hash(const Ch* b, const Ch* e) const {return do_hash(b,e);}
  string_type transform(const Ch* b, const Ch* e) const {return do_transform(b,e);}
```

```
    static locale::id id;    // объект идентификации фасета (§D.2, §D.3, §D.3.1)
  protected:
    ~collate();              // внимание: protected деструктор

    virtual int do_compare(const Ch* b, const Ch* e, const Ch* b2, const Ch* e2)
  const;
    virtual string_type do_transform(const Ch* b, const Ch* e) const;
    virtual long do_hash(const Ch* b, const Ch* e) const;
  };
```

Как и любые фасеты, ***collate*** наследуется от ***facet*** и предоставляет конструктор с аргументом, указывающим, отвечает ли класс ***locale*** за время жизни фасета (§D.3).

Обратите внимание на то, что деструктор объявляется в секции ***protected***. Фасет ***collate*** не предназначен для непосредственного использования, а служит базовым классом для наследования от него конкретных классов сравнения строк под управлением ***locale*** (§D.3). В рамках интерфейса ***collate*** прикладные программисты и разработчики библиотек могут писать свои собственные фасеты сравнения строк.

Функция ***compare***() выполняет базовую операцию сравнения строк в рамках правил, установленных для конкретного ***collate***; она возвращает ***1***, если первая строка лексикографически больше второй; возвращает ***0***, если строки идентичны, и возвращает ***−1*** в случае, когда вторая строка больше первой. Например:

```
void f(const string& s1, const string& s2, collate<char>& cmp)
{
    const char* cs1 = s1.data();    // поскольку compare() работает с char[]
    const char* cs2 = s2.data();

    switch(cmp.compare(cs1, cs1+s1.size(), cs2, cs2+s2.size()))
    {
      case 0:                       // строки идентичны согласно cmp
        // ...
        break;
      case -1 :                     // s1 < s2
        // ...
        break;
      case 1:                       // s1 > s2
        // ...
        break;
    }
}
```

Обратите внимание, что функции-члены ***collate*** сравнивают массивы элементов типа ***Ch***, а не строки ***basic_string*** или С-строки с терминальным нулем. В частности, числовое значение ***0*** для ***Ch*** воспринимается как обычный символ, а не как терминальный нуль. Кроме того, ***compare***() отличается от ***strcmp***() тем, что возвращает именно ***1***, ***0*** и ***−1***, а не нуль и произвольные положительные и отрицательные значения (§20.4.1).

Стандартный библиотечный тип ***string*** не чувствителен к локализациям — он выполняет сравнение строк по правилам для набора символов реализации (§C.2). Более того, стандартный тип ***string*** не позволяет непосредственно задать критерий сравнения (глава 20). Чтобы выполнить сравнение, чувствительное к локализации,

мы можем использовать функцию ***compare***() фасета ***collate***. С точки зрения простоты записи, лучше использовать ***compare***() неявно, через функцию-операцию ***operator***() () класса ***locale*** (§D.2.4). Например:

```
void f(const string& s1, const string& s2, const char* n)
{
  bool b = s1 == s2;

  const char* cs1 = s1.data();   // поскольку compare() работает с char[]
  const char* cs2 = s2.data();

  typedef collate<char> Col;

  const Col& glob = use_facet<Col>(locale());       // из текущей глобаль. локализации
  int i0= glob.compare(cs1,cs1+s1.size(),cs2,cs2+s2.size());

  const Col& my_coll = use_facet<Col>(locale("")); // из предпочтительной локал-ции
  int i1 = my_coll.compare(cs1,cs1+s1.size(),cs2,cs2+s2.size());

  const Col& coll = use_facet<Col>(locale(n));     // из locale с именем n
  int i2 = coll.compare(cs1,cs1+s1.size(),cs2,cs2+s2.size());

  int i3 = locale()(s1,s2);      // сравнение при текущей глобальной локализации
  int i4 = locale("")(s1,s2);    // сравнение при предпочтительной локализации
  int i5 = locale(n)(s1,s2);     // сравнение при локализации с именем n
  // ...
}
```

Здесь ***i0==i3***, ***i1==i4*** и ***i2==i5***. Нетрудно представить случаи, когда ***i2***, ***i3*** и ***i4*** различаются. Рассмотрим последовательность слов из немецкого словаря:

Dialekt, Diät, dich, dichten, Dichtung

По соглашению, существительные (и только) пишутся с заглавной буквы, но их порядок не зависит от регистра букв.

Зависящая от регистра букв сортировка расположила бы эти слова в таком порядке:

Dialekt, Diät, Dichtung, dich, dichten

Буква ***ä*** (***a*** умляут) интерпретируется как «нечто вроде ***a***», и поэтому она стоит раньше буквы ***c***. Однако в большинстве символьных кодовых наборов числовое значение для ***a*** больше числового значения для ***c***, то есть ***int***(***ä***) >***int***(***c***), и на выходе простой сортировки на базе числовых значений будет:

Dialekt, Dichtung, Diät, dich, dichten

Написание функции сравнения, которая корректно упорядочивает данную последовательность слов точно по словарю, является интересным упражнением (§D.6[3]).

Функция ***hash***() вычисляет *хэш-значение* (§17.6.2.3), которое требуется для построения *хэш-таблиц*.

Функция ***transform***() производит из исходной строки некую строку, сравнение с которой третьих строк дает тот же результат, что и их сравнение с исходной строкой. Применение этой функции оптимизирует код, в котором некоторые строки сравниваются с одной и той же строкой. Это характерно для фрагментов, в которых, например, некоторая строка ищется во множестве других строк.

Функции ***compare***(), ***hash***() и ***transform***(), объявленные в секции ***public***, в своей реализации вызывают защищенные виртуальные функции ***do_compare***(), ***do_hash***() и ***do_transform***(), соответственно; последние могут замещаться в производных классах. Такая двухслойная стратегия позволяет разработчикам библиотек при реализации невиртуальных функций обеспечить некоторую общую функциональность для всех вызовов в независимости от того, что будет реализовано в «do-функциях».

Использование виртуальных функций обеспечивает, хотя и несколько накладным образом, полиморфную природу фасетов. Во избежание лишних накладных расходов на эти вызовы ***locale*** может установить точную природу используемого фасета и кэшировать любые значения, необходимые для повышения производительности (§D.2.2).

Статический член ***id*** типа ***locale::id*** используется для идентификации фасета (§D.3). Стандартные функции ***has_facet*** и ***use_facet*** полагаются на соответствие этих идентификаторов и фасетов (§D.3.1). Два фасета, предоставляющие локализации один и тот же интерфейс и семантику, должны иметь одно и то же значение ***id***. Например, у ***collate<char>*** и ***collate_byname<char>*** эти идентификаторы одинаковые. И наоборот, два фасета, выполняющие разные функции по отношению к локализации, должны иметь разные ***id***. Например, у ***numpunct<char>*** и у ***num_put<char>*** они разные (§D.4.2).

D.4.1.1. Именованные фасеты сравнения

Фасет ***collate_byname*** является для конкретной локализации вариацией ***collate*** с именем, заданным строковым аргументом конструктора:

```
template<class Ch>
class std::collate_byname: public collate<Ch>
{
public:
  typedef basic_string<Ch> string_type;

  // конструирование из именованной локализации:
  explicit collate_byname(const char*, size_t r = 0);

  // внимание: ни нового id, ни новых функций

protected:
  ~collate_byname();        // внимание: protected деструктор

  // замещаем виртуальные функции collate<Ch>:
  int do_compare(const Ch* b, const Ch* e, const Ch* b2, const Ch* e2) const;
  string_type do_transform(const Ch* b, const Ch* e) const;
  long do_hash(const Ch* b, const Ch* e) const;
};
```

Таким образом, фасетом ***collate_byname*** можно пользоваться для извлечения фасета ***collate*** из локализации, именованной в рамках системной среды выполнения программы (§D.4). Очевидным способом хранения фасетов в среде выполнения является запись необходимых данных в файл на диске. Менее гибкой альтернативой будет представление фасета в виде кода и данных в фасете с суффиксом ***_byname***.

Класс ***collate_byname<char>*** является примером фасета, у которого нет собственного ***id*** (§D.3). В рамках ***locale*** фасеты ***collate_byname<Ch>*** и ***collate<Ch>*** взаимозаменяемы. Фасеты ***collate*** и ***collate_byname*** для одной и той же локализации различаются лишь дополнительным конструктором фасета ***collate_byname***, а также его семантикой.

Отметим, что деструктор у ***collate_byname*** защищенный. Это означает, что нельзя определить локальные (в функции) объекты типа ***collate_byname***:

```
void f()
{
  collate_byname<char> my_coll(""); // error: невозможно уничтожить my_coll
  // ...
}
```

Это отражает взгляд на то, что контексты локализации и фасеты лучше определять где-нибудь на более высоком уровне, чтобы они могли оказывать влияние на более обширные фрагменты программ. Например, установка глобального контекста (§D.2.3) или прикрепление контекста локализации к потоку (§21.6.3, §D.1). Но при необходимости, мы можем наследовать от ***collate_byname*** класс с открытым деструктором и создавать локальные объекты такого класса.

D.4.2. Ввод и вывод чисел

Вывод чисел выполняется фасетом ***num_put*** посредством записи в буфер потока (§21.6.4). Ввод чисел выполняется фасетом ***num_get*** путем чтения из буфера потока. Формат, используемый фасетами ***num_put*** и ***num_get***, задается *«фасетом пунктуации чисел»* — ***numpunct***.

D.4.2.1. Пунктуация чисел

Фасет ***numpunct*** определяет формат ввода/вывода встроенных типов, таких как ***bool***, ***int*** и ***double***:

```
template<class Ch>
class std::numpunct: public locale::facet
{
public:
  typedef Ch char_type;
  typedef basic_string<Ch> string_type;

  explicit numpunct(size_t r = 0);

  Ch decimal_point() const;        // '.' в classic()
  Ch thousands_sep() const;        // ',' в classic()
  string grouping() const;         // "" в classic() - значит, что нет группировки

  string_type truename() const;    // "true" в classic()
  string_type falsename() const;   // "false" в classic()

  static locale::id id;            // объект идентификации фасета (§D2, §D.3, §D.3.1)

protected:
  ~numpunct();
  // виртуальные "do_"-функции (см. §D.4.1)
};
```

Символы строки, возвращаемые функцией *grouping*(), читаются как последовательность небольших целых значений, каждое из которых идентифицирует число цифр в группе. Символ с номером *0* соответствует самой правой группе (то есть наименее значимым цифрам), символ с номером *1* — группу слева от предыдущей и т.д. Таким образом, "*\004\002\003*" описывает формат чисел вида *123-45-6789* (при условии, что в качестве разделителя используется знак '-'). При необходимости последнее числовое значение в образце группировки может быть повторено. В нашем примере "*\003*" эквивалентно, например, "*\003\003\003*". Само название функции *thousands_sep*() (разделение тысяч), возвращающей символ разделения, говорит о том, что основное предназначение группировки заключается в том, чтобы представить большие числа в более читаемой форме. Функции *grouping*() и *thousands_sep*() определяют формат и для ввода, и для вывода целых чисел. Они также определяют формат целой части чисел с плавающей запятой, но не цифр после *decimal_point*().

Можно определить новый стиль пунктуации чисел, создавая производный от *numpunct* класс. Например, я могу определить фасет *My_punct* для записи целых значений тройками цифр с пробелом между ними, и для записи чисел с плавающей запятой с использованием запятой в Европейском стиле, отделяющей дробную часть от целой части числа:

```
class My_punct: public std::numpunct<char>
{
public:
  typedef char char_type;
  typedef string string_type;

  explicit My_punct(size_t r = 0)  : std::numpunct<char>(r)  { }

protected:
  char do_decimal_point() const {return ',';}   // запятая
  char do_thousands_sep() const {return ' ';}   // пробел
  string do_grouping() const {return "\003";}   // 3-цифровые группы
};

void f()
{
  cout << setprecision(4) << fixed;
  cout<< "style A: " << 12345678<< " *** " << 1234.5678 << '\n';

  locale loc(locale(), new My_punct);
  cout.imbue(loc);
  cout<< "style B: " << 12345678<< " *** " << 1234.5678 << '\n';
}
```

На выходе получим:

```
style A: 12345678 *** 1234.5678
style B: 12 345 678 *** 1 234,5678
```

Заметьте, что *imbue*() хранит копию своего аргумента в своем потоке. Следовательно, поток может использовать прикрепленную к нему локализацию даже после того, как ее оригинал будет уничтожен. Если *iostream* имеет установленный флаг *boolalpha* (§21.2.2, §21.4.1), то для символьного представления *true* и *false* использу-

ются возвраты функций ***truename***() и ***falsename***(), соответственно; в противном случае используются ***1*** и ***0***.

Имеется и соответствующая ***_byname*** версия для ***numpunct*** (§D.4, §D.4.1):

```
template<class Ch> class std::numpunct_byname: public numpunct<Ch> { /*...*/ };
```

D.4.2.2. Вывод чисел

При записи в буфер потока (§21.6.4) ***ostream*** использует фасет ***num_put***:

```
template<class Ch, class Out = ostreambuf_iterator<Ch> >
class std::num_put: public locale::facet
{
public:
  typedef Ch char_type;
  typedef Out iter_type;

  explicit num_put(size_t r = 0);

  // поместить значение "v" в позицию буфера "b" потока "s":
  Out put(Out b, ios_base& s, Ch fill, bool v) const;
  Out put(Out b, ios_base& s, Ch fill, long v) const;
  Out put(Out b, ios_base& s, Ch fill, unsigned long v) const;
  Out put(Out b, ios_base& s, Ch fill, double v) const;
  Out put(Out b, ios_base& s, Ch fill, long double v) const;
  Out put(Out b, ios_base& s, Ch fill, const void* v) const;

  static locale::id id;           // объект идентификации фасета (§D.2, §D.3,§D.3.1)

protected:
  ~num_put();
  // виртуальные "do_"-функции (см. §D.4.1)
};
```

Аргумент ***Out*** (итератор вывода; §19.1, §19.2.1) идентифицирует место в буфере потока ***ostream*** (§21.6.4), в которое ***put***() помещает символы, представляющие выводимое числовое значение. Возврат функции ***put***() — это итератор, указывающий на позицию, следующую за последним записанным символом.

Отметим, что умолчательная специализация ***num_put*** (та, чей итератор для доступа к символам имеет тип ***ostreambuf_iterator<Ch>***) является частью стандартного контекста локализации (§D.4). Если вам нужна иная специализация, придется создать ее самостоятельно. Например:

```
template<class Ch>
class String_numput: public std::num_put<Ch,typename basic_string<Ch>::iterator>
{
public:
  String_numput() : std::num_put<Ch, typename basic_string<Ch>::iterator>(1) { }
};

void f(int i, string& s, int pos)      // форматирование i в s с позиции pos
{
  String_numput<char> f;
  ios_base& xxx = cout;                 // используем правила форматирования потока cout
  f.put(s.begin()+pos,xxx,' ',i);      // форматируем i в s
}
```

Аргумент ***ios_base*** используется для получения информации о состоянии форматирования и локализации. Например, если требуется осуществить заполнение определенного пространства, используется символ ***fill***, как того требует аргумент ***ios_base***. Как правило, буфер потока, в который пишут через итератор ***b***, является буфером, ассоциированным с ***ostream***, для которого ***s*** является базовым классом. Заметим, что сконструировать объект типа ***ios_base*** не просто. Он, в частности, управляет множеством аспектов форматирования, которые нужно согласовывать с целью достижения приемлемого вывода. Соответственно, в ***ios_base*** нет открытого конструктора (§21.3.3).

Функция ***put***() также пользуется аргументом ***ios_base*** для доступа к локализации потока. Она используется для определения пунктуации (§D.4.2.1), символьного представления булевых значений и для преобразований к ***Ch***. Например, если ***s*** является аргументом типа ***ios_base*** у функции ***put***(), то в исходном тексте этой функции можно было бы встретить следующие фрагменты кода:

```
const locale& loc = s.getloc();
//...
wchar_t w = use_facet<ctype<char> >(loc).widen(c);                // из char в Ch
//...
string pnt = use_facet<numpunct<char> >(loc).decimal_point();  // по умолчанию: '.'
//...
string flse = use_facet<numpunct<char> >(loc).falsename();      // по умолчанию: "false"
```

Стандартные фасеты, такие как ***num_put<char>***, обычно используются неявно в функциях стандартного ввода/вывода; поэтому большинству программистов нет необходимости знать о них. Тем не менее, использование таких фасетов стандартными библиотечными функциями представляет интерес, поскольку иллюстрирует работу потоков ввода/вывода и способы обращения с фасетами. Как всегда, стандартная библиотека демонстрирует примеры интересных программных технологий.

Используя ***num_put***, разработчик ***ostream*** мог бы написать:

```
template<class Ch, class Tr>
ostream& std::basic_ostream<Ch,Tr>::operator<<(double d)
{
  sentry guard(*this);      // см. §21.3.8
  if(!guard) return *this;

  try
  {
    if(use_facet<num_put<Ch> >(getloc()).put(*this,*this,this->fill(),d).failed())
      setstate(badbit);
  }
  catch(...)
  {
    handle_ioexception(*this);
  }
  return *this;
}
```

В этом примере много интересного. Класс ***sentry*** (часовой) обеспечивает гарантию выполнения всех префиксных и постфиксных операций (§21.3.8). Мы получаем контекст локализации потока ***ostream*** вызовом его функции-члена ***getloc***()

(§21.7). Извлекаем ***num_put*** из ***locale*** при помощи ***use_facet*** (§D.3.1). После этого мы обращаемся к соответствующей функции ***put***() для выполнения реальной работы. Итератор ***ostreambuf_iterator*** можно сконструировать из ***ostream*** (§19.2.6), а ***ostream*** можно неявно преобразовать к его базовому классу ***ios_base*** (§21.2.1), чтобы обеспечить два первых аргумента для функции ***put***().

Функция ***put***() возвращает свой аргумент, являющийся итератором вывода. Этот итератор вывода извлекается из ***basic_ostream***, так что он имеет тип ***ostreambuf_iterator***. Следовательно, для проверки корректности состояния и для установки такого состояния нам доступна функция ***failed***() (§19.2.6.1).

Я не использую ***use_facet***, поскольку гарантируется, что стандартные фасеты (§D.4) присутствуют в любом контексте локализации. Если эта гарантия нарушается, генерируется исключение ***bad_cast*** (§D.3.1).

Функция ***put***() вызывает виртуальную функцию ***do_put***(), так что может быть задействован пользовательский код и ***operator***<<() должен быть готов к обработке исключения, генерируемого замещающим вариантом ***do_put***(). Кроме того, ***num_put*** может отсутствовать для некоторых символьных типов, так что ***use_facet***() может сгенерировать исключение ***std::bad_cast*** (§D.3.1). Поведение операции << для встроенных типов, таких как ***double***, определено стандартом языка C++. Стало быть вопрос не в том, что должна делать функция ***handle_ioexception***(), а в том, как она должна действовать в соответствии со стандартом. Если в исключении у ostream установлен флаг ***badbit*** (§21.3.6), то исключение просто повторно генерируется. В противном случае обработка исключения сводится к установке состояния потока и продолжению работы. В обоих случаях нужно установить флаг ***badbit*** (§21.3.3):

```
template<class Ch, class Tr>
void handle_ioexception(std::basic_ostream<Ch,Tr>& s)    // вызывается из catch-блока
{
  if(s.exceptions() & ios_base::badbit)
  {
    try
    {
      s.setstate(ios_base::badbit);                // может генер-вать basic_ios::failure
    }
    catch(...) {}
    throw;                                         // повторная генерация исключения
  }
  s.setstate(ios_base::badbit);
}
```

Здесь ***try***-блок необходим, поскольку ***setstate***() может сгенерировать исключение ***basic_ios::failure*** (§21.3.3, §21.3.6). Однако если в исключительном состоянии потока установлен флаг ***badbit***, тогда ***operator***<<() должен повторно сгенерировать исключение, приведшее к вызову ***handle_ioexception***() (а не просто сгенерировать ***basic_ios::failure***).

Реализация операции << для встроенных типов, таких как ***double***, должна писать непосредственно в буфер потока. Реализуя операцию << для пользовательских типов, мы можем избежать ненужной сложности путем сведения вывода пользовательских типов к выводу уже существующих типов (§D.3.2).

D.4.2.3. Ввод чисел

При чтении из буфера потока (§21.6.4) ***istream*** полагается на фасет ***num_get***:

```
template<class Ch, class In = istreambuf_iterator<Ch> >
class std::num_get: public locale::facet
{
public:
  typedef Ch char_type;
  typedef In iter_type;

  explicit num_get(size_t r = 0);

  // читаем [b:e) в v по правилам форматирования из s; об ошибках - через r:
  In get(In b, In e, ios_base& s, ios_base::iostate& r, bool& v) const;
  In get(In b, In e, ios_base& s, ios_base::iostate& r, long& v) const;
  In get(In b, In e, ios_base& s, ios_base::iostate& r, unsigned short& v) const;
  In get(In b, In e, ios_base& s, ios_base::iostate& r, unsigned int& v) const;
  In get(In b, In e, ios_base& s, ios_base::iostate& r, unsigned long& v) const;
  In get(In b, In e, ios_base& s, ios_base::iostate& r, float& v) const;
  In get(In b, In e, ios_base& s, ios_base::iostate& r, double& v) const;
  In get(In b, In e, ios_base& s, ios_base::iostate& r, long double& v) const;
  In get(In b, In e, ios_base& s, ios_base::iostate& r, void*& v) const;

  static locale::id id;     // объект идентификации фасета (§D.2, §D.3, §D.3.1)

protected:
  ~num_get();
  // виртуальные "do_"-функции (см. §D.4.1)
};
```

В основном ***num_get***() реализован так же, как и ***num_put*** (§D.4.2.2). Поскольку ***get***() читает, а не пишет, то ей нужна пара итераторов ввода, а аргумент, обозначающий целевой объект для читаемых данных, представлен ссылкой.

Для уведомления о состоянии потока устанавливается переменная ***r*** типа ***iostate***. Если значение желаемого типа прочитать не удается, то флаг ***failbit*** в переменной ***r*** устанавливается; если был достигнут конец ввода, то в ***r*** устанавливается eofbit. Операция ввода использует ***r*** для того, чтобы решить, как установить состояние своего потока. Если ошибок не обнаружено, то считанное значение присваивается ***v***; в противном случае переменная ***v*** остается неизменной.

Класс ***sentry*** (часовой) обеспечивает гарантию выполнения всех префиксных и постфиксных операций потока (§21.3.8). В частности, применение ***sentry*** гарантирует, что мы начинаем читать, лишь если поток находится в надлежащем для чтения состоянии.

Разработчик ***istream*** может написать следующее:

```
template<class Ch, class Tr>
istream& std::basic_istream<Ch,Tr>::operator>>(double& d)
{
  sentry guard(*this);              // см. §21.3.8
  if(!guard) return *this;

  iostate state = 0;                // все хорошо
  istreambuf_iterator<Ch> eos;
  double dd;
```

```
    try
    {
      use_facet<num_get<Ch> >(getloc()).get(*this, eos, *this, state, dd);
      if(state==0 || state==eofbit) d = dd;
      setstate(state);
    }
    catch(...)
    {
      handle_ioexception(*this);        // см. §D.4.2.2
    }
    return *this;
}
```

В случае ошибки функция ***setstate***() сгенерирует исключения, определенные для ***istream*** (§21.3.6).

Определив ***numpunct***, такой как, например, ***My_punct*** из §D.4.2, мы можем читать с использованием нестандартной пунктуации. Например:

```
void f()
{
    cout << "style A: ";
    int i1;
    double d1;
    cin >> i1 >> d1;                // читаем по стандартному формату "12345678"

    locale loc(locale::classic(), new My_punct);
    cin.imbue(loc);
    cout << "style B: ";
    int i2;
    double d2;
    cin >> i1 >> d2;                // читаем по формату "12 345 678"
}
```

Чтобы читать по-настоящему необычные числовые форматы, нам нужно заместить ***do_get***(). Например, мы могли бы определить ***num_get*** для чтения римских цифр вроде ***XXI*** или ***MM*** (§D.6[15]).

D.4.3. Ввод и вывод финансовой информации

Форматирование количества денег технически аналогично форматированию «просто чисел» (§D.4.2). Однако представление денежных величин еще более чувствительно к национальным стандартам и особенностям. Например, отрицательные значения (убыток, дебет), таких как ***–1.25***, должно в некоторых контекстах представляться положительными числами, заключенными в круглые скобки: (***1.25***). Аналогично, в других контекстах могут применять цвет для наглядного отображения отрицательных сумм денег.

Нет никакого стандартного «денежного типа». Вместо этого предполагается, что программист явным образом применит «денежные фасеты» к числовым величинам, про которые он знает, что они соответствуют денежным суммам. Например:

```
class Money
{
  long int amount;

public:
  Money (long int i) : amount (i) { }
  operator long int () const {return amount; }
};
//...
void f (long int i)
{
  cout << "value= " << i << " amount= " <<Money (i) << endl;
}
```

Денежные фасеты призваны упростить написание операции вывода для ***Money***, которая выводила бы денежные суммы в соответствии с национальными особенностями (соглашениями) (§D.4.3.2). Результат вывода будет зависеть от того, какой ***locale*** закреплен за ***cout***. Вот возможные варианты форматов вывода:

```
value= 1234567 amount= $12345.67
value= 1234567 amount= 12345,67 DKK
value= -1234567 amount= $-12345.67
value= -1234567 amount= -$12345.67
value= -1234567 amount= (CHF12345,67)
```

Для денежных сумм считается важным учет, по возможности, самых малых монет. Поэтому я принял соглашение о представлении целым значением количества центов (пенсов, центов, копеек и т.д.), а не долларов (фунтов, евро, рублей и т.д.). Это соглашение поддерживается посредством функции ***frac_digits***() фасета ***moneypunct*** (§D.4.3.1). Вид «десятичной точки» определяется при помощи ***decimal_point***().

Фасеты ***money_get*** и ***money_put*** предоставляют функции, которые выполняют ввод/вывод на основе формата, определяемого фасетом ***money_base***.

Наш простой тип ***Money*** может использоваться для управления форматами ввода/вывода или хранения денежных величин. В первом случае мы приводим к типу ***Money*** иные типы, хранящие денежные суммы, непосредственно перед их записью, и читаем в переменные типа ***Money*** перед их конвертированием в другие типы. Во втором случае мы исходно храним денежные суммы в переменных типа ***Money***, что менее чревато ошибками, ибо нет необходимости выполнять преобразования при записи в потоки, и не надо преобразовывать прочитанные данные к форматам, учитывающим национальные особенности. Но, к сожалению, не всегда можно внедрить тип ***Money*** глубоко в программы, которые не разрабатывались с этой целью. Остается выполнять преобразования к типу ***Money*** в операциях ввода/вывода.

D.4.3.1. Пунктуация денежных величин

Фасет ***moneypunct***, управляющий представлением денежных величин, похож, естественно, на фасет ***numpunct***, управляющий представлением «обычных чисел» (§D.4.2.1):

```
class std::money_base
{
public:
  enum part {none, space, symbol, sign, value};   // части раскладки значения
  struct pattern { char field[4]; };             // спецификация раскладки
};

template<class Ch, bool International = false>
class std::moneypunct: public locale::facet, public money_base
{
public:
  typedef Ch char_type;
  typedef basic_string<Ch> string_type;

  explicit moneypunct(size_t r = 0);

  Ch decimal_point() const;            // '.' в classic()
  Ch thousands_sep() const;            // ',' в classic()
  string grouping() const;             // "" в classic() - означает "нет группировки"

  string_type curr_symbol() const;     // "$" в classic()
  string_type positive_sign() const;   // "" в classic()
  string_type negative_sign() const;   // "-" в classic()

  int frac_digits() const;        // число цифр после десятичной точки; 2 в classic()

  pattern pos_format() const;          // { symbol, sign, none, value } в classic()
  pattern neg_format() const;          // { symbol, sign, none, value } в classic()

  static const bool intl = International;  // применяем международ. денеж. форматы

  static locale::id id;           // объект идентификации фасета (§D.2, §D.3,§D.3.1)

protected:
  ~moneypunct();
  // виртуальные "do_"-функции (см. §D.4.1)
};
```

Средства, которые предлагает фасет ***moneypunct,*** предназначены, в первую очередь, для разработчиков реализаций фасетов ***money_put*** и ***money_get*** (§D.4.3.2, §D.4.3.3).

Функции-члены ***decimal_point***(), ***thousands_sep***() и ***grouping***() ведут себя так же, как их эквиваленты из ***numpunct.***

Функции-члены ***curr_symbol***(), ***positive_sign***() и ***negative_sign***() возвращают строку, изображающую валюту (например, ***$***, ***FRF***, ***DKK*** и т.д.), плюсовой знак и минусовой знак, соответственно. Если аргумент ***International*** шаблона равен ***true***, то член ***intl*** также будет равен ***true***, что повлечет за собой использование международных обозначений валютных символов. Такие международные обозначения являются четырехсимвольными строками. Например:

```
"USD"
"DKK"
"EUR"
```

Последним символом является терминальный нуль. Стандарт ISO-4217 определяет трехбуквенные идентификаторы валют. Когда ***International*** равен ***false***, можно использовать локальные обозначения валют, вроде ***$*** и т.д.

Объект типа ***pattern***, возвращаемый функциями ***pos_format***() или ***neg_format***(), состоит из четырех частей, определяющих последовательность, в которой отображаются числовое значение, валютный символ, символ знака (плюс или минус) и заполнители (пробельные символы). Наиболее общие форматы задаются тривиально с применением этого простого образца (***pattern***) формата. Например:

```
+$123.45       // { sign, symbol, space, value } где positive_sign() возвращает "+"
$+123.45       // { symbol, sign, value, none } где positive_sign() возвращает "+ "
$123.45        // { symbol, sign, value, none } где positive_sign() возвращает ""
$123.45-       // { symbol, value, sign, none }
-123.45 DKK    // { sign, value, space, symbol}
($123.45)      // { sign, symbol, value, none } где negative_sign() возвращает "()"
(123.45DKK)    // { sign, value, symbol, none } где negative_sign() возвращает "()"
```

Представление отрицательных значений с использованием скобок достигается, когда возврат функции ***negative_sign***() содержит пару символов (). Первый символ этой строки помещается туда, где часть ***sign*** располагается в образце, а остальная часть строки — после всех остальных частей образца. Наиболее распространенным применением этого средства является принятое в финансовых кругах заключение отрицательных денежных сумм в круглые скобки, но возможны и иные варианты применения. Например:

```
-$123.45         // {sign,symbol,value,none} где negative_sign() возвращает "–"
*$123.45 silly   // {sign,symbol,value,none} где negative_sign() возвращает "* silly"
```

Каждое из значений ***sign***, ***value*** и ***symbol*** должно появиться в образце ровно один раз. Оставшееся значение может быть ***space*** или ***none***. В том месте, где в образце расположен ***space***, в отображаемом значении может появиться один или более пробельных символов. А где расположен ***none*** (но не в конце образца) — там может быть нуль или более пробельных символов.

Обратите внимание на то, что эти строгие правила запрещают такие на первый взгляд разумные образцы, как

```
pattern pat = {sign, value, none, none};    // error: нету symbol
```

Функция ***frac_digits***() указывает, где помещается ***decimal_point***(). Часто денежные суммы представляются в самых мелких валютных единицах (§D.4.3). Такая единица обычно в сто раз меньше основной валютной единицы (доллара, фунта, рубля и т.д.), и поэтому ***frac_digits***() чаще всего возвращает 2.

Приведем пример простого формата, оформленного в виде фасета:

```
class My_money_io : public moneypunct<char, true>
{
public:
  explicit My_money_io(size_t r = 0) : moneypunct<char, true>(r) { }

  char_type do_decimal_point() const {return '.';}
  char_type do_thousands_sep() const {return ',';}
  string do_grouping() const {return "\003\003\003";}

  string_type do_curr_symbol() const {return "USD";}
  string_type do_positive_sign() const {return "";}
  string_type do_negative_sign() const {return "()";}
```

```
int do_frac_digits() const {return 2;}      // две цифры после десятичной точки
pattern do_pos_format() const
{
  static pattern pat = {sign, symbol, value, none};
  return pat;
}
pattern do_neg_format() const
{
  static pattern pat = {sign, symbol, value, none};
  return pat;
}
};
```

Этот фасет используется в операциях ввода/вывода ***Money***, определенных в §D.4.3.2 и §D.4.3.3.

Для moneypunct имеется и версия с суффиксом ***_byname*** (§D.4, §D.4.1):

```
template<class Ch, bool Intl = false>
class std::moneypunct_byname: public moneypunct<Ch,Intl> {/*...*/};
```

D.4.3.2. Вывод денежных величин

Фасет ***money_put*** записывает денежные суммы в соответствии с форматом, заданным moneypunct. Более точно, ***money_put*** предоставляет функции ***put***(), которые помещают надлежащим образом отформатированное представление символов в буфер потока:

```
template<class Ch, class Out = ostreambuf_iterator<Ch> >
class std::money_put: public locale::facet
{
public:
  typedef Ch char_type;
  typedef Out iter_type;
  typedef basic_string<Ch> string_type;

  explicit money_put(size_t r = 0);

  // поместить значение "v" в позицию буфера "b":
  Out put(Out b, bool intl, ios_base& s, Ch fill, long double v) const;
  Out put(Out b, bool intl, ios_base& s, Ch fill, const string_type& v) const;

  static locale::id id;      // объект идентификации фасета (§D.2, §D.3, §D.3.1)

protected:
  ~money_put();
  // виртуальные "do_"-функции (см. §D.4.1)
};
```

Аргументы ***b***, ***s***, ***fill*** и ***v*** используются так же, как в функциях ***put***() фасета ***num_put*** (§D.4.2.2). Аргумент ***intl*** указывает, используется ли стандартный международный четырехсимвольный знак валюты, или же с этой целью используется какой-нибудь «местный» символ (§D.4.3.1).

Исходя из ***money_put***, мы можем определить операцию вывода для ***Money*** (§D.4.3):

```
ostream& operator<< (ostream& s, Money m)
{
  ostream::sentry guard(s);                    // см. §21.3.8
  if(!guard) return s;

  try
  {
    const money_put<char>& f = use_facet<money_put<char> >(s.getloc());
    if(m==static_cast<long double>(m))   // m можно представить как long double
    {
      if(f.put(s,true,s,s.fill(),m).failed()) s.setstate(ios_base::badbit);
    }
    else
    {
      ostringstream v;
      v << m;                               // преобразуем в строковое представление
      if(f.put(s,true,s,s.fill(),v.str()).failed()) s.setstate(ios_base::badbit);
    }
  }
  catch(...)
  {
    handle_ioexception(s);                 // см. §D.4.2.2
  }
  return s;
}
```

Если для точного представления денежных сумм не хватает типа ***long double***, я преобразовываю значение в его строковое представление и вывожу это функцией ***put***(), принимающей строковый аргумент.

D.4.3.3. Ввод денежных величин

Фасет ***money_get*** читает денежные суммы в соответствии с форматом, заданным с помощью moneypunct. Более точно, ***money_get*** предоставляет функции ***get***(), которые извлекают надлежащим образом отформатированное представление символов из буфера потока:

```
template<class Ch, class In = istreambuf_iterator<Ch> >
class std::money_get: public locale::facet
{
public:
  typedef Ch char_type;
  typedef In iter_type;
  typedef basic_string<Ch> string_type;

  explicit money_get(size_t r = 0);

  // читаем [b:e) в v по форматным правилам из s; ошибки - устанавливаем r:
  In get(In b, In e, bool intl, ios_base& s, ios_base::iostate& r, long double& v) const;
  In get(In b, In e, bool intl, ios_base& s, ios_base::iostate& r, string_type& v) const;

  static locale::id id;     // объект идентификации фасета (§D.2, §D.3, §D.3.1)

protected:
  ~money_get();
  // виртуальные "do_"-функции (см. §D.4.1)
};
```

Аргументы ***b***, ***e***, ***s***, ***fill*** и ***v*** используются так же, как в функциях ***get***() фасета ***num_get*** (§D.4.2.3). Аргумент ***intl*** указывает, используется ли стандартный международный четырехсимвольный знак валюты, или же с этой целью используется какой-нибудь «местный» символ (§D.4.3.1).

Правильно определенная пара фасетов ***money_get*** и ***money_put*** обеспечит вывод в форме, которая может быть считана обратно без ошибок и потери информации. Например:

```
int main ()
{
  Money m;
  while (cin>>m) cout << m << "\n";
}
```

Вывод этой программы может быть принят ею в качестве ввода. Более того, вывод, полученный в результате второго запуска этой программы с вводом, равным выводу первого запуска, должен быть идентичен исходному вводу.

Операция ввода для ***Money*** может выглядеть следующим образом:

```
istream& operator>> (istream& s, Money& m)
{
  istream::sentry guard (s) ;                       // см. §21.3.8
  if (guard)
   try
   {
     ios_base::iostate state = 0;                   // все хорошо
     istreambuf_iterator<char> eos;
     string str;

     use_facet<money_get<char> > (s.getloc ()) .get (s, eos, true, state, str) ;

     if (state==0 || state==ios_base::eofbit)
     {
       long int i = strtol (str.c_str () , 0, 0) ;  // про strtol() см. §20.4.1
       if (errno==ERANGE)
         state |= ios_base::failbit;
       else
         m = i;
       s.setstate (state) ;
     }
   }
   catch (...)
   {
     handle_ioexception (s) ;                       // см. §D.4.2.2
   }
   return s;
}
```

Здесь использована функция ***get***(), которая читает в строковую переменную, поскольку чтение в переменную типа ***double*** с последующим преобразованием в ***long int*** может привести к потере точности.

D.4.4. Ввод и вывод дат и времени

К сожалению, стандартная библиотека C++ не предоставляет удовлетворительного типа для даты. Из стандартной библиотеки языка C она получает по наследству низкоуровневые средства работы с датами и интервалами времени. Эти средства языка C являются фундаментом для средств C++, позволяющим работать со временем удобным образом, не зависящим от системных особенностей.

В следующих разделах показывается, как представление дат и времени дня можно сделать чувствительными к контексту локализации. Приводится пример того, как пользовательский тип *Date* может быть вписан в архитектуру потоков (глава 21) и контекстов локализации (§D.2). Реализация *Date* демонстрирует технологии, полезные при обращении со временем в том случае, если у вас нет типа *Date*.

D.4.4.1. Часы и таймеры

На самом низком уровне большинство систем имеет таймер, отсчитывающий малые интервалы времени. Стандартная библиотека предоставляет функцию *clock*(), которая возвращает значение зависящего от реализации арифметического типа *clock_t*. Возврат функции *clock*() может быть откалиброван макросом *CLOCKS_PER_SEC*. Если у вас нет доступа к надежной утилите измерения времени, можно написать свою собственную измерительную программу:

```
int main (int argc, char* argv[])    // §6.1.7
{
  int n = atoi(argv[1]);             // §20.4.1
  clock_t t1 = clock();

  if(t1 == clock_t(-1))              // clock_t(-1) означает, что clock() не работает
  {
    cerr << "sorry, no clock\n";
    exit(1);
  }

  for(int i = 0; i<n; i++) do_something();   // измерительный цикл
  clock_t t2 = clock()
  if(t2 == clock_t(-1))
  {
    cerr << "sorry, clock overflow\n";
    exit(2);
  }

  cout << "do_something() "<<n<<" times took "
       << double(t2-tl) /CLOCKS_PER_SEC << " seconds"
       << " (measurement granularity: " << CLOCKS_PER_SEC << " of a second)\n";
}
```

Явное преобразование *double*(*t2-t1*) перед операцией деления необходимо, поскольку значение *clock_t* может быть целым. Точное значение, когда начинает работать функция *clock*(), зависит от реализации; предполагается, что *clock*() измеряет интервалы времени в течение одного запуска программы. Для величин *t1* и *t2*, возвращаемых функцией *clock*(), выражение *double*(*t2-t1*) / *CLOCKS_PER_SEC* является наилучшей оценкой в секундах для времени между двумя вызовами.

Если для конкретного процессора таймер не реализован, или если интервал времени слишком велик для измерения, то **clock()** возвращает значение **clock_t(-1)**.

Функция **clock()** предназначена для измерения интервалов времени длительностью от долей секунды до нескольких секунд. Например, если тип **clock_t** есть 32-битовый знаковый **int** и **CLOCKS_PER_SEC** равно 1000000, то можно использовать **clock()** для измерения в микросекундах интервалов от 0 до 2000 секунд (порядка получаса).

Обратите внимание, что получение доверительных результатов измерения времени работы программы может оказаться нетривиальным делом. Другие исполняемые на машине программы могут влиять на это время; кэш и взаимодействие между процессами являются труднопрогнозируемыми, да и алгоритмы обработки могут зависеть от данных неожиданным образом. Если вы уж пытаетесь замерить что-либо, то хотя бы делайте это несколько раз и отбрасывайте сильно отличающиеся результаты как недостоверные.

Для работы с более продолжительными интервалами и календарным временем, стандартная библиотека предоставляет тип **time_t** для представления момента времени и структуру **tm** для "препарирования" этого момента:

```
typedef implementation_defined time_t; // зависящий от реализации арифметический тип
                                        // (§4.1.1), чаще всего это 32-bit целое

struct tm
{
  int tm_sec;     // секунда минуты [0,61]; 60 и 61 - корректировочные секунды
  int tm_min;     // минута часа [0,59]
  int tm_hour;    // час дня [0,23]
  int tm_mday;    // день месяца [1,31]
  int tm_mon;     // месяц года [0,11]; 0 значит January
  int tm_year;    // год после 1900; 0 значит 1900, а 102 значит 2002
  int tm_wday;    // день после Sunday [0,6]; 0 значит Sunday
  int tm_yday;    // день после January 1 [0,365]; 0 значит January 1
  int tm_isdst;   // часы летнего времени
};
```

Отметим, что стандарт гарантирует содержание в **tm** указанных здесь целочисленных полей. Стандарт не гарантирует, что они расположены именно в указанном порядке, или же что нет других полей.

Типы **time_t** и **tm** вместе со стандартными средствами их использования представлены в <**ctime**> и <**time.h**>. К примеру:

```
clock_t clock();                    // кол-во "тиков" с момента начала работы программы

time_t time(time_t* pt);            // текущее календарное время
double difftime(time_t t2, time_t t1); // t2-t1 в секундах

tm* localtime(const time_t* pt);    // местное время для *pt
tm* gmtime{const time_t* pt);       // GMT (Greenwich Mean Time) для *pt, или 0
                                    // (официально называется Coordinated Universal Time,UTC)

time_t mktime(tm* ptm);             // time_t для *ptm, или time_t(-1)
char* asctime(const tm* ptm);       // представление *ptm в стиле С-строк
                                    // например, "Sun Sep 16 01:03:52 1973\n"

char* ctime(const time_t* t) {return asctime(localtime(t));}
```

Будьте осторожны — как *localtime()*, так и *gmtime()* возвращают указатель *tm** на статически размещенный объект, так что последовательные вызовы функции изменяют значение этого объекта. Так что, либо немедленно используйте возвращаемое значение, либо скопируйте *tm* в управляемую вами область памяти. Аналогично, *asctime()* возвращает указатель на статически размещенный массив символов.

Структура *tm* позволяет представить даты в диапазоне по крайней мере десятков тысяч лет (около [-32000,32000] для минимального размера типа *int*). В то же время, *time_t* — это чаще всего 32-битовое *long int*, что позволяет типу *time_t* представлять в секундах диапазон длиною в 68 лет в обе стороны от базового года. Базовым годом обычно является год 1970; точным базовым временем при этом является 0:00, 1 января по Гринвичу. Если же *time_t* — это 32-битное *int*, то уже к 2038 году мы окажемся «вне времени», если, конечно, не перейдем на более длинные *int*, как это уже сделано на некоторых системах.

Тип *time_t* предназначен для работы с относительно близкими датами, так что нельзя гарантировать его работу вне диапазона [1902,2038]. Еще хуже то, что не все реализации функций для работы со временем одинаково работают с отрицательными значениями. Для переносимости требуется, чтобы значение, которое нужно представлять и как *time_t*, и как *tm*, лежало в диапазоне [1902,2038]. Для представления дат вне этого диапазона нужен какой-либо иной механизм.

Одним из следствий вышесказанного является тот факт, что функция *mktime()* может отработать некорректно. Для аргументов, которые не представимы типом *time_t*, она возвращает индикатор ошибки *time_t(-1)*.

Хронометраж в программах с большим временем работы можно выполнить следующим образом:

```
int main(int argc, char* argv[])      // §6.1.7
{
  time_t t1 = time(0);
  do_a_lot(argc, argv);
  time_t t2 = time(0);
  double d = difftime(t2, t1);
  cout << "do_a_lot() took" << d << " seconds\n";
}
```

Если аргумент-указатель функции *time()* не *0*, то результирующее время записывается и по этому указателю. Если календарное время не реализовано (например, на специализированных процессорах), то возвращается значение *time_t(-1)*. Можно попытаться определить текущую дату следующим образом:

```
int main()
{
  time_t t;
  if(time(&t) == time_t(-1))          // time_t(-1) означает, что time() не работает
  {
    cerr << "Bad time\n";
    exit(1);
  }
  tm* gt = gmtime(&t);
  cout << gt->tm_mon+1 << '/' << gt->tm_mday << '/' << 1900+gt->tm_year<<endl;
}
```

D.4.4.2. Класс Date

Как упоминалось в §10.3, маловероятно, чтобы единственный тип ***Date*** мог удовлетворить все потребности. Различные способы использования дат диктуют разные способы их представления, а календарная информация до 19 столетия сильно зависит от капризов истории. Тем не менее, в качестве примера мы можем определить тип ***Date*** примерно как в §10.3, используя для реализации ***time_t***:

```
class Date
{
public:
  enum Month {jan=1, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec};

  class Bad_date {};

  Date(int dd, Month mm, int yy);
  Date();

  friend ostream& operator<<(ostream& s, const Date& d);
  // ...

private:
  time_t d;                          // стандартное представление даты и времени
};

Date::Date(int dd, Month mm, int yy)
{
  tm x = {0};
  if(dd<0 || 31<dd) throw Bad_date();   // переупрощено: см. §10.3.1
  x.tm_mday = dd;
  if(mm<jan || dec<mm) throw Bad_date();
  x.tm_mon = mm-1;                      // tm_mon отсчитывается с нуля
  x.tm_year = yy-1900;                  // tm_year отсчитывается от 1900
  d = mktime(&x);
}

Date::Date()
{
  d = time(0);                          // умолчательный Date: сегодня
  if(d == time_t(-1)) throw Bad_date();
}
```

Теперь перед нами встает задача определить операции << и >> для класса ***Date***, которые были бы чувствительны к локализации.

D.4.4.3. Вывод дат и времени

Как и ***num_put*** (§D.4.2), фасет ***time_put*** предоставляет функции ***put()*** для записи в буферы через итераторы:

```
template<class Ch, class Out = ostreambuf_iterator<Ch> >
class std::time_put: public locale::facet
{
public:
  typedef Ch char_type;
  typedef Out iter_type;
```

```
    explicit time_put (size_t r = 0) ;

    // помещаем t в буфер потока s через b по формату fmt:
    Out put (Out b, ios_base& s, Ch fill, const tm* t,
            const Ch* fmt_b, const Ch* fmt_e) const;
    Out put (Out b, ios_base& s, Ch fill, const tm* t, char fmt, char mod = 0) const
    {return do_put (b,s,fill,t,fmt,mod) ; }

    static locale::id id;     // объект идентификации фасета (§D.2, §D.3, §D.3.1)

protected:
    ~time_put () ;
    virtual Out do_put (Out, ios_base&, Ch, const tm*, char, char) const;
};
```

Вызов **put** (*b,s,fill,t,fmt_b,fmt_e*) помещает информацию о дате из *t* в буфер потока *s* через *b*. Символ *fill* используется в качестве заполняющего. Формат вывода специфицируется *printf*()-подобной строкой [*fmt_b,fmt_e*). Этот *printf*()-подобный формат (§21.8) используется для порождения настоящего вывода и может содержать следующие форматные спецификаторы

- %a сокращение для дня недели (например, Sat)
- %A полное название дня недели (например, Saturday)
- %b сокращенное название месяца (например, Feb)
- %B полное название месяца (например, February)
- %c дата и время (например, Sat Feb 06 21:46:05 1999)
- %d число [01,31] (например, 06)
- %H 24-часовой формат времени [00,23] (например, 21)
- %I 12-часовой формат времени [01,12] (например, 09)
- %j день года [001,366] (например, 037)
- %m месяц [01,12] (например, 02)
- %M минута часа [00,59] (например, 48)
- %p a.m/p.m индикатор для 12-часового формата (например, PM)
- %S секунда в минуте [00,61] (например, 40)
- %U неделя года [00,53] начиная с воскресенья (например, 05)
- %w день недели [0,6], где 0 это воскресенье (например, 6)
- %W неделя года [00,53] с понедельника (например, 05)
- %x дата (например, 02/06/99)
- %X время (например, 21:48:40)
- %y год без века [00,99] (например, 99)
- %Y год (например, 1999)
- %Z индикатор часового пояса (например, EST), если он известен

Этот длинный список форматных спецификаторов может послужить наглядным аргументом в пользу расширяемой системы ввода/вывода. Однако как и всякая специализированная система обозначений она точна и даже, до некоторой степени, удобна.

Кроме перечисленных директив форматирования многие реализации поддерживают «модификаторы», такие как, например, целые модификаторы ширины поля вывода (§21.8), *%10X*. Модификаторы форматов дат и времени не являются частью стандарта C++, в то время как стандарты некоторых платформ, например POSIX, требуют их. Поэтому избежать применения модификаторов трудно, несмотря на то, что их применение снижает переносимость программы.

Функция *strftime*() из <*ctime*> или <*time.h*>, подобная функции *sprintf*() (§21.8), производит вывод в соответствии с директивами форматирования дат и времени:

```
size_t strftime(char* s, size_t max, const char* format, const tm* tmp);
```

Эта функция помещает максимум *max* символов из **tmp* в **s* согласно формату *format*. Например:

```
int main()
{
  const int max = 20;   // надеемся, что strftime() не произведет более 20 символов
  char buf[max];

  time_t t = time(0);
  strftime(buf, max, "%A\n", localtime(&t));
  cout << buf;
}
```

Для дня недели «среда» эта программа выведет *Wednesday* в контексте локализации «по умолчанию» *classic*() (§D.2.3), и *onsdag* для датской локализации.

Символы, не являющиеся частью формата, такие как «перевод строки» в данном примере, просто копируются в **s*.

Когда *put*() обнаруживает символ форматирования *f* (и необязательный модификатор *m*), то для выполнения форматирования она вызывает виртуальную функцию *do_put*() с соответствующими параметрами: *do_put*(*b*, *s*, *fill*, *t*, *f*, *m*).

Вызов *put*(*b*, *s*, *fill*, *t*, *f*, *m*) является упрощенным вариантом *put*(), в котором символ форматирования *f* и модификатор *m* задаются явным образом. Таким образом, фрагмент

```
const char fmt[] = "%10X";
put(b,s,fill,t,fmt,fmt+sizeof(fmt));
```

можно сократить до

```
put(b,s,fill,t,'X',10);
```

Если формат содержит мультибайтные символы, он должен и начинаться, и заканчиваться в состоянии «по умолчанию» (§D.4.6).

Мы можем применить функцию *put*(), чтобы реализовать зависящую от контекста локализации операцию вывода для типа *Date*:

```
ostream& operator<<(ostream& s, const Date& d)
{
  ostream::sentry guard(s);          // см. §21.3.8
  if(!guard) return s;

  tm* tmp = localtime(&d.d);
  try
```

```
  {
    if(use_facet<time_put<char> >(s.getloc()).put(s,s,s.fill(),tmp,'x').failed())
      s.setstate(ios_base::failbit);
  }
  catch(...)
  {
    handle_ioexception(s);          // см. §D.4.2.2
  }
  return s;
}
```

Поскольку никакого стандартного типа ***Date*** не существует, нет и стандартного формата «по умолчанию» для ввода/вывода дат. Здесь я задал формат `%x`, передавая символ `'x'` в качестве символа форматирования. Поскольку формат `%x` является форматом «по умолчанию» для ***get_time***`()` (§D.4.4.4), то он является по сути дела «наиболее стандартным».

Имеется и версия ***time_put*** с суффиксом ***_byname*** (§D.4, §D.4.1):

```
template<class Ch, class Out = ostreambuf_iterator<Ch> >
class std::time_put_byname: public time_put<Ch,Out> { /* ... */ };
```

D.4.4.4. Ввод дат и времени

Как всегда, ввод несколько сложнее вывода. Когда мы пишем код для вывода значения, у нас обычно есть выбор между несколькими разными форматами. А когда мы пишем код для ввода, мы должны в первую очередь следить за ошибками и, лишь иногда, выбирать из нескольких форматных альтернатив.

Фасет ***time_get*** реализует ввод дат и времени. Идея заключается в том, что фасет ***time_get*** может читать форматы времени и дат, которые в данной локализации порождает фасет ***time_put***. Ввиду отсутствия стандартных классов времени и дат, программист в рамках контекста локализации может осуществлять вывод согласно множеству форматов. Например, все приведенные ниже представления могут быть получены с помощью одной и той же операции вывода, использующей фасет ***time_put*** (§D.4.4.5) разных локализаций:

January 15th 1999
Thursday 15th January 1999
15 Jan 1999AD
Thurs 15/1/99

Стандарт C++ поощряет разработчиков ***time_get*** воспринимать форматы дат и времени в соответствии с POSIX и иными распространенными промышленными стандартами. Главная трудность состоит в том, что почти невозможно стандартизировать намерение читать даты и время во всех мыслимых форматах, принятых в рамках разнообразных местных и национальных (шире — культурных) соглашений. Можно поэкспериментировать, чтобы поточнее познакомиться с возможностями конкретного контекста локализации (§D.6[8]). Если формат не воспринимается, программист может реализовать необходимый ему альтернативный фасет ***time_get*** сам.

Стандартный фасет для ввода времени, ***time_get***, наследуется от класса ***time_base***:

```
struct std::time_base
{
  enum dateorder
  {
    no_order,          // не упорядочено
    dmy,               // день-месяц-год
    mdy,               // месяц-день-год
    ymd,               // год-месяц-день
    ydm                // год-день-месяц
  };
};
```

Разработчик может воспользоваться этим перечислением для упрощения анализа форматов даты.

Подобно ***num_get***, фасет ***time_get*** осуществляет доступ к своему буферу посредством пары итераторов ввода:

```
template<class Ch, class In = istreambuf_iterator<Ch> >
class time_get: public locale::facet, public time_base
{
public:
  typedef Ch char_type;
  typedef In iter_type;

  explicit time_get(size_t r = 0);

  dateorder date_order() const {return do_date_order();}

  // читаем [b,e) в d по форматным правилам из s; ошибки - установка r:
  In get_time(In b, In e, ios_base& s, ios_base::iostate& r, tm* d) const;
  In get_date(In b, In e, ios_base& s, ios_base::iostate& r, tm* d) const;
  In get_year(In b, In e, ios_base& s, ios_base::iostate& r, tm* d) const;

  In get_weekday(In b, In e, ios_base& s, ios_base::iostate& r, tm* d) const;
  In get_monthname(In b, In e, ios_base& s, ios_base::iostate&.r, tm* d) const;

  static locale::id id;      // объект идентификации фасета (§D.2, §D.3, §D.3.1)

protected:
  ~time_get();
  // виртуальные "do_"-функции (см. §D.4.1)
};
```

Функция ***get_time***`()` вызывает ***do_get_time***`()`. По умолчанию ***get_time***`()` читает время так, как оно выводится функцией ***time_put::put***`()` в рамках контекста локализации, используя формат `%`***x*** (§D.4.4).

Таким образом, простейший вариант операции ввода для типа ***Date*** выглядит примерно так:

```
istream& operator>>(istream& s, Date& d)
{
  istream::sentry guard(s);        // см. §21.3.8
  if(!guard) return s;

  ios_base::iostate res = 0;
  tm x = {0};
  istreambuf_iterator<char,char_traits<char> > end;
```

```
try
{
  use_facet<time_get<char> > (s.getloc()).get_date(s,end,s,res,&x);
  if(res==0 || res==ios_base::eofbit)
    d = Date(x.tm_mday,Date::Month(x.tm_mon+1),x.tm_year+1900);
  else
    s.setstate(res);
}
catch(...)
{
  handle_ioexception(s);      // см. §D.4.2.2
}
return s;
}
```

Вызов ***get_date(s, end, s, res, &x)*** полагается на два неявных приведения типа от типа ***istream***: первый аргумент ***s*** используется для создания ***istreambuf_iterator***; третий аргумент ***s*** конвертируется к типу ***ios_base*** (базовый класс для ***istream***).

Эта операция ввода будет работать корректно для дат из диапазона значений, представимых с помощью ***time_t***. Вот несложный тестировочный фрагмент:

```
int main()
try
{
  Date today;
  cout << today << endl;                  // пишем в формате %x
  Date d(12,Date::may,1998);

  cout << d << endl;
  Date dd;
  while(cin >> dd) cout << dd << endl;    // читаем даты в формате %x
}
catch(Date::Bad date)
{
  cout << "exit: bad date caught\n";
}
```

Имеется и версия ***time_get*** с суффиксом ***_byname*** (§D.4, §D.4.1):

```
template<class Ch, class In = istreambuf_iterator<Ch> >
class std::time_get_byname: public time_get<Ch,In> {/* ... */};
```

D.4.4.5. Более гибкий класс Date

Если вы попробуете использовать класс ***Date*** из §D.4.4.2 с вводом/выводом из §D.4.4.3 и §D.4.4.4, то вы обнаружите его ограниченность:

1. Он работает только с датами, представимыми с помощью типа ***time_t***, что в типичном случае означает диапазон [1970,2038].
2. Он воспринимает даты только в стандартном формате — каков бы он ни был.
3. Его реакция на ошибки ввода неприемлема.
4. Он поддерживает лишь потоки ***char*** (а не произвольных символьных типов).

Более интересная и полезная операция ввода должна воспринимать более широкий диапазон дат, распознавать несколько распространенных форматов и надежным образом реагировать на ошибки ввода. Для этого нужно уйти от использования типа ***time_t***:

```
class Date
{
public:
  enum Month {jan=1,feb,mar,apr,may,jun,jul,aug,sep,oct,nov,dec};

  struct Bad_date
  {
    const char* why;
    Bad_date(const char* p) : why(p) { }
  };

  Date(int dd, Month mm, int yy, int day_of_week = 0);
  Date();

  void make_tm(tm* t) const;         // поместить tm представление Date в *t
  time_t make_time_t() const;        // возврат time_t представления Date

  int year() const {return y;}
  Month month() const {return m;}
  int day() const {return d;}
  // ...

private:
  char d;
  Month m;
  int y;
};
```

Для простоты я вернулся к представлению ***(d,m,y)*** (§10.2).

Конструктор можно определить следующим образом:

```
Date::Date(int dd, Month mm, int yy, int day_of_week)
    : d(dd), m(mm), y(yy)
{
  if(d==0 && m==Month(0) && y==0) return;         // Date(0,0,0) - "нулевая дата"
  if(mm<jan || dec<mm) throw Bad_date("bad month");

  if(dd<1 || 31<dd)                                // переупрощено; см. §10.3.1
   throw Bad_date("bad day of month");

  if(day_of_week && day_in_week(yy,mm,dd) != day_of_week)
   throw Bad_date("bad day of week");
}

Date::Date() : d(0), m(0), y(0) { }                // "нулевая дата"
```

Алгоритм ***day_in_week()*** весьма нетривиален и не имеет непосредственного отношения к локализациям, так что я отставил его в сторону. Если он вам потребуется, поищите его в рамках вашей системы.

Операции сравнения всегда полезны для типов вроде ***Date***:

```
bool operator== (const Date& x, const Dat& y)
{
  return x.year()==y.year() && x.month()==y.month() && x.day()==y.day();
}

bool operator!= (const Date& x, const Date& y)
{
  return !(x==y);
}
```

Поскольку мы расстались со стандартными типами ***tm*** и ***time_t***, нам нужны будут операции преобразования к этим типам для обеспечения взаимодействия с программами, ожидающими эти типы:

```
void Date::make_tm(tm* p) const
{
  tm x = {0};
  *p = x;
  p->tm_year = y-1900;
  p->tm_mday = d;
  p->tm_mon = m-1;
}

time_t Date::make_time_t() const
{
  if(y<1970 || 2038<y)                  // переупрощено
    throw Bad_date("date out of range for time_t");

  tm x;
  make_tm(&x);
  return mktime(&x);
}
```

D.4.4.6. Задание формата даты

Язык C++ не определяет стандартного формата вывода дат (*%x* является всего лишь наиболее вероятным претендентом; §D.4.4.3). Однако даже если бы и существовал стандартный формат, нам все равно потребовались бы альтернативы. Это можно реализовать, предоставив формат по умолчанию и способ его замены. Например:

```
class Date_format
{
  static char fmt[];                 // умолчательный формат
  const char* curr;                  // текущий формат
  const char* curr_end;

public:
  Date_format() : curr(fmt), curr_end(fmt+strlen(fmt)) { }

  const char* begin() const {return curr;}
  const char* end() const {return curr_end;}

  void set(const char* p, const char* q) {curr=p; curr_end=q;}
  void set(const char* p) {curr=p; curr_end=curr+strlen(p);}

  static const char* default_fmt() {return fmt;}
};
```

```
const char Date_format::fmt[] = "%A, %B%d, %Y";//например, Friday, February 5 1999
Date_format date_fmt;
```

Чтобы иметь возможность использовать формат ***strftime***() (§D.4.4.3), я воздержался от параметризации класса ***Date_format*** по типу используемых символов. Таким образом, данное решение допускает только те форматы дат, которые представимы с помощью ***char***[]. Кроме того, я воспользовался глобальным объектом форматирования ***date_fmt*** для представления формата ***Date*** по умолчанию. Поскольку значение ***date_fmt*** можно изменять, то мы получаем грубый способ управления форматом ***Date***, аналогичный применению ***global***() (§D.2.3).

Более общим решением было бы определение фасетов ***Date_in*** и ***Date_out*** для управления чтением и записью в потоки. Этот подход представлен в §D.4.4.7.

Располагая ***Date_format***, можно написать операцию ***Date***::***operator***<<() следующим образом:

```
template<class Ch, class Tr>
basic_ostream<Ch,Tr>& operator<<(basic_ostream<Ch,Tr>& s, const Date& d)
// запись согласно пользовательскому формату
{
  typename basic_ostream<Ch,Tr>::sentry guard(s);    // см. §21.3.8
  if(!guard) return s;

  tm t;
  d.make_tm(&t);

  try
  {
    const time_put<Ch>& f = use_facet<time_put<Ch> >(s.getloc());
    if(f.put(s,s,s.fill(),&t,date_fmt.begin(),date_fmt.end()).failed())
      s.setstate(ios_base::failbit);
  }
  catch(...)
  {
    handle_ioexception(s);                           // см. §D.4.2.2
  }
  return s;
}
```

Я мог бы воспользоваться ***use_facet*** для проверки того, что фасет ***time_put<Ch>*** имеется в контексте локализации потока ***s***. Однако в данном случае проще выглядит вариант с перехватом исключения, генерируемого из ***use_facet***.

Приведем простую тестовую программу, управляющую форматом вывода с помощью ***date_fmt***:

```
int main()
try
{
  while(cin >> dd && dd != Date()) cout << dd << endl; // исполь-ем умолчат. date_fmt

  date_fmt.set("%Y/%m/%d");
  while(cin >> dd && dd != Date()) cout << dd << endl; // используем "%Y/%m/%d"
}
catch(Date::Bad_date e)
{
```

```
  cout << "bad date caught: " << e.why << endl;
}
```

D.4.4.7. Фасет ввода даты

Как всегда, ввод немного сложнее вывода. Но поскольку интерфейс доступа к низкоуровнему вводу ограничен функцией ***get_date***() и поскольку операция ввода ***operator>>***(), определенная для ***Date*** в §D.4.4.4, не имеет прямого доступа к представлению ***Date***, мы можем воспользоваться этой операцией без изменений. Вот параметризованная версия операции ввода, соответствующая операции вывода ***operator<<***() из §D.4.4.6:

```
template<class Ch, class Tr>
istream<Ch,Tr>& operator>>(istream<Ch,Tr>& s, Date& d)
{
  typename istream<Ch,Tr>::sentry guard(s);

  if(guard)
   try
   {
    ios_base::iostate res = 0;
    tm x = {0};
    istreambuf_iterator<Ch,Tr> end;

    use_facet<time_get<Ch> >(s.getloc()).get_date(s,end,s,res,&x);

    if(res==0 || res==ios_base::eofbit)
      d =
Date(x.tm_mday,Date::Month(x.tm_mon+1),x.tm_year+1900,x.tm_wday);
    else
      s.setstate(res);
   }
  catch(...)
  {
    handle_ioexception(s);     // см. §D.4.2.2
  }
  return s;
}
```

Эта операция ввода для типа ***Date*** вызывает функцию ***get_date***() из фасета ***time_get*** (§D.4.4.4) для потока ***istream***. Поэтому можно обеспечить иную и более гибкую форму ввода, определив новый фасет, производный от ***time_get***:

```
template<class Ch, class In = istreambuf_iterator<Ch> >
class Date_in: public std::time_get<Ch,In>
{
public:
  Date_in(size_t r = 0) : std::time_get<Ch>(r) { }

protected:
  In do_get_date(In b, In e, ios_base& s, ios_base::iostate& r, tm* tmp) const;

private:
  enum Vtype {novalue,unknown,dayofweek,month};
  In getval(In b, In e, ios_base& s, ios_base::iostate& r, int* v, Vtype* res) const;
};
```

Функция ***getval***() должна прочитать год, месяц и день (месяца), а также, возможно, и день недели, после чего составить из этого ***tm***.

Названия месяцев и дней недели зависят от конкретного контекста локализации. Следовательно, мы не можем упоминать их непосредственно в нашей функции ввода. Вместо этого мы будем распознавать месяцы и дни посредством вызова функций, которые фасет ***time_get*** предоставляет для этой цели: ***get_monthname***() и ***get_weekday***() (§D.4.4.4).

Год, день месяца и, возможно, месяц представляются целыми числами. К сожалению, само число не указывает, относится ли оно к месяцу, дню или к чему-либо еще. Например, 7 может означать июль месяц, или 7-е число, или даже 2007 год. Настоящей целью функции ***date_order***() из фасета ***time_get*** является разрешение таких двусмысленностей.

Стратегия фасета ***Date_in*** состоит в том, чтобы читать значения, классифицировать их и вызывать ***date_order***(), чтобы узнать, имеют ли смысл (и какой) введенные значения. Закрытая функция ***getval***() осуществляет непосредственное чтение буфера потока ввода и начальную классификацию:

```
template<class Ch, class In>
In Date_in<Ch,In>::getval(In b, In e, ios_base& s, ios_base::iostate& r,
                          int* v, Vtype* res) const
// Читаем части Date: число, день недели, или месяц.
// Пропускаем пробелы и пунктуацию.
{
  const ctype<Ch>& ct = use_facet<ctype<Ch> >(s.getloc()); // ctype - см. §D.4.5
  Ch c;

  *res = novalue;                  // значение не найдено

  for(;;)                          // пропускаем пробелы и пунктуацию
  {
    if(b == e) return e;
    c = *b;
    if(!(ct.is(ctype_base::space,c) || ct.is(ctype_base::punct,c))) break;
    ++b;
  }

  if(ct.is(ctype_base::digit,c))   // читаем целое, не обращая внимание на numpunct
  {
    int i = 0;

    do  // переводим цифру из произвольного символьного типа в десятичное значение:
    {
      static char const digits[] = "0123456789";
      i = i*10 + find(digits,digits+10,ct.narrow(c,' ')) -digits;
      c= *++b;
    }
    while(ct.is(ctype_base::digit,c));

    *v = i;
    *res = unknown;     // целое значение, но неизвестно, чему оно соответствует
    return b;
  }
```

```
    if(ct.is(ctype_base::alpha ,c))          // поиск названия месяца или дня недели
    {
      basic_string<Ch> str;
      while(ct.is(ctype_base::alpha,c))      // читаем символы в строку
      {
        str += c;
        if(++b == e) break;
        c = *b;
      }

      tm t;
      basic_stringstream<Ch> ss(str);
      typedef istreambuf_iterator<Ch> SI;          // итераторный тип для буфера ss
      get_monthname(ss.rdbuf(),SI(),s,r, &t);  // читаем из буфера потока в памяти
      if((r&(ios_base::badbit|ios_base::failbit))==0)
      {
        *v= t.tm_mon;
        *res = month;
        return b;
      }
      r = 0;                                  // очистка состояния перед повторным чтением
      get_weekday(ss.rdbuf(),SI(),s,r, &t);       // читаем из буфера потока в памяти
      if((r&(ios_base::badbit|ios_base::failbit))==0)
      {
        *v = t.tm_wday;
        *res = dayofweek;
        return b;
      }
    }
    r |= ios_base::failbit;
    return b;
  }
```

Главный трюк здесь заключается в том, чтобы отличить месяцы от дней недели. Мы читаем с помощью итераторов ввода, так что мы не можем дважды прочитать интервал [*b*, *e*), сначала отыскивая месяц, а затем день недели. С другой стороны, мы не можем принять решение, глядя лишь на один символ, потому что только функции ***get_monthname***() и ***get_weekday***() знают, какая последовательность символов означает название месяца и дня недели в данной локализации. Я принял решение читать строки алфавитных символов в тип ***string***, создать поток ***stringstream*** из ***string***, после чего вычитывать уже ***streambuf*** этого потока.

Обработка ошибок состоит в непосредственном установлении битов состояния, таких как ***ios_base*::*badbit***. Это необходимо из-за того, что более удобные функции манипулирования состоянием потока, такие как ***clear***() и ***set_state***(), определены в ***basic_ios***, а не в его базовом классе ***ios_base*** (§21.3.3). При необходимости операция >> использует информацию об ошибках, получаемую от ***get_date***(), для переустановки состояния потока ввода.

Располагая функцией ***getval***(), мы можем сначала прочитать значения, и лишь после этого проверять, имеют ли они смысл. Функция ***date_order***() критически важна:

```
template<class Ch, class In>
In Date_in<Ch,In>::do_get_date(In b, In e, ios_base& s, ios_base::iostate& r,
                            tm* tmp) const
// необязательный день недели, за которым идут ymd, dmy, mdy, или ydm
{
  int val[3];                       // для day, month и year в некотором порядке
  Vtype res[3] = {novalue};         // для классификации значений

  for(int i=0; b!=e && i<3; ++i)    // читаем day, month и year
  {
    b = getval(b,e,s,r,&val[i],&res[i]);
    if(r) return b;                 // oops: error
    if(res[i]==novalue)             // незавершенная дата
    {
      r |= ios_base::badbit;
      return b;
    }
    if(res[i]==dayofweek)
    {
      tmp->tm_wday = val[i];
      --i;                          // oops: не day, month или year
    }
  }

  time_base::dateorder order = date_order();  // попытаемся придать смысл
                                              // прочитанным значениям

  if(res[0]==month)                           // mdy или error
  {
    // ...
  }
  else if(res[1]==month)                      // dmy или ymd или error
  {
    tmp->tm_mon = val[1];
    switch (order)
    {
      case dmy:
         tmp->tm_mday = val[0];
         tmp->tm_year = val[2};
         break;
      case ymd:
         tmp->tm_year = val[0];
         tmp->tm_mday = val[2];
         break;
      default:
         r |= ios_base::badbit;
         return b;
    }
  }
  else if(res[2]==month)                      // ydm или error
  {
    // ...
  }
```

```
    else                                    // полагаемся на dateorder или error
    {
      // ...
    }

    tmp->tm_year -= 1900;                   // подгоняем базовый год под tm-соглашения
    return b;
}
```

Я опустил часть кода, которая не вносит ничего нового в понимание контекстов локализации, дат, или же обработки ввода. Оставляем в качестве упражнения написание более универсальных функций ввода дат (§D.6[9-10]).

Вот простая тестовая программа:

```
int main ()
try
{
  cin.imbue (loc (locale (), new Date_in)) ;      // читаем даты, используя Date_in
  while (cin >> dd && dd != Date ()) cout << dd << endl;
}
catch (Date::Bad_date e)
{
  cout << "bad date caught: " << e.why << endl;
}
```

Заметьте, что ***do_get_date*()** примет и бессмысленные даты, такие как

Thursday October 7, 1998

и

1999/Feb/31

Проверка согласованности года, месяца, числа и, возможно, дня недели, выполняется в конструкторе ***Date***. Именно класс ***Date*** должен знать, что является корректной датой, а ***Date_in*()** не обязан разделять это знание.

Можно написать ***getval*()** или ***do_get_date*()** таким образом, чтобы они пытались угадывать смысл числовых значений. Например, очевидно, что

12 May 1922

не означает 1922 число 12 года. То есть мы можем догадаться. что число, которое не может быть месяцем, является годом. Подобного рода догадки бывают полезными в конкретных ограниченных смысловых контекстах. Но в более общих случаях польза от таких догадок сомнительна. Например:

12 May 15

может соответствовать любому году из следующих: 12, 15, 1912, 1915, 2012 или 2015. Часто бывает полезным снабжать числа подсказками. Например, ***1-ое*** и ***15-ое*** — это дни, а "***751 до н.э.***" и "***1453 н.э.***" — это годы.

D.4.5. Классификация символов

При чтении символов часто возникает задача их классификации с целью придания смысла прочитанному. Например, чтобы прочесть число, процедура ввода должна знать, какие символы являются цифрами. Это похоже на пример из §6.1.2, который демонстрирует классификацию символов стандартными классифицирующими функциями с целью разбора вводимых выражений.

Естественно, классификация символов зависит от используемого алфавита. Как следствие, фасет *ctype* используется для классификации символов в рамках контекста локализации.

Символы классифицируются в соответствии с перечислением ***mask***:

```
class std::ctype_base
{
public:
  enum mask                    // реальные значения зависят от реализации
  {
    space = 1,                 // пробельные символы (в "C" locale: ' ','\n','\t', ...)
    print = 1<<1,              // символы печати
    cntrl = 1<<2,              // управляющие символы
    upper = 1<<3,              // символы в верхнем регистре
    lower = 1<<4,              // символы в нижнем регистре
    alpha = 1<<5,              // алфавитные символы
    digit = 1<<6,              // десятичные цифры
    punct = 1<<7,              // символы пунктуации
    xdigit = 1<<8,             // шестнадцатеричные цифры
    alnum=alpha | digit,       // алфавитно-цифровые символы
    graph=alnum | punct
  };
};
```

Тип ***mask*** не зависит от конкретного типа символов. Как следствие, это перечисление помещается в нешаблонный базовый класс.

Очевидно, что ***mask*** отражает традиционную для C и C++ классификацию (§20.4.1). Однако для разных наборов символов, разные символы попадут в разные группы. Например, для набора ASCII целое значение 125 соответствует символу '}', означающему знак пунктуации (***punct***). А в датском национальном наборе символов этот числовой код соответствует гласной 'å', которая в датской локализации должна классифицироваться как ***alpha***.

Данная классификация получила название mask (маска), поскольку традиционной эффективной реализацией для классификации небольших наборов символов является таблица, в которой каждый вход (строка, элемент) хранит битовое представление классификации. Например:

```
table['a'] == lower | alpha | xdigit
table['1'] == digit
table[' '] == space
```

При такой реализации ***table***[***c***]&***m*** отлично от нуля, если символ ***c*** есть ***m***, и ***0*** в противном случае.

Фасет ***ctype*** определяется следующим образом:

```
template<class Ch>
class std::ctype: public locale::facet, public ctype_base
{
public:
  typedef Ch char_type;
  explicit ctype(size_t r = 0);

  bool is(mask m, Ch c) const;                                       // "c" есть "m"?

  // поместить классификацию каждого Ch из [b:e) в v:
  const Ch* is(const Ch* b, const Ch* e, mask* v) const;

  const Ch* scan_is(mask m, const Ch* b, const Ch* e) const;     // найти m,
  const Ch* scan_not(mask m, const Ch* b, const Ch* e) const;    // найти не-m

  Ch toupper(Ch c) const;
  const Ch* toupper(Ch* b, const Ch* e) const;                   // convert [b:e)
  Ch tolower(Ch c) const;
  const Ch* tolower(Ch* b, const Ch* e) const;

  Ch widen(char c) const;
  const char* widen(const char* b, const char* e, Ch* b2) const;
  char narrow(Ch c, char def) const;
  const Ch* narrow(const Ch* b, const Ch* e, char def, char* b2) const;

  static locale::id id;     // объект идентификации фасета (§D.2, §D3, §D.3.1)

protected:
  ~ctype();
  // виртуальные "do_"-функции (см. §D.4.1)
};
```

Вызов *is(m, c)* проверяет, принадлежит ли символ *c* классификации *m*. Например:

```
int count_spaces(const string& s, const locale& loc)
{
  const ctype<char>& ct = use_facet<ctype<char> >(loc);
  int i = 0;

  for(string::const_iterator p = s.begin(); p != s.end(); ++p)
    if(ct.is(ctype_base::space, *p)) ++i;     // символ пробела в соответствии с ct

  return i;
}
```

Обратите внимание, что *is()* можно использовать для проверки принадлежности символа к одной из нескольких классификаций. Например:

```
ct.is(ctype_base::space|ctype_base::punct, c); // с в ct -пробел.сим-л или знак пунктуации?
```

Вызов *is(b, e, v)* выявляет классификацию каждого символа из *[b, e)* и помещает ее в соответствующую позицию массива *v*.

Вызов *scan_is(m, b, e)* возвращает указатель на первый символ из *[b, e)*, который есть *m*. Если нет символов, относящихся к классификации *m*, то возвращается *e*. Как всегда для стандартных фасетов, открытые функции-члены реализованы путем вызова своих виртуальных "do-функций". Простая реализация может выглядеть следующим образом:

```
template<class Ch>
const Ch* std::ctype<Ch>::do_scan_is(mask m, const Ch* b, const Ch* e) const
{
 while(b!=e && !is(m,*b)) ++b;
 return b;
}
```

Вызов *scan_not*(*m*, *b*, *e*) возвращает указатель на первый символ из [*b*, *e*), который не есть *m*. Если все символы относятся к классификации *m*, то возвращается *e*.

Вызов *toupper*(*c*) возвращает *c* в верхнем регистре, если это допустимо в используемом наборе символов; или сам *c* в противном случае.

Вызов toupper(*b*, *e*) конвертирует каждый символ из интервала [*b*, *e*) в верхний регистр и возвращает *e*. Простая реализация может выглядеть следующим образом:

```
template<class Ch>
const Ch* std::ctype<Ch>::to_upper(Ch* b, const Ch* e)
{
 for(; b!=e; ++b) *b = toupper(*b);
 return e;
}
```

Функция *tolower*() аналогична функции *toupper*(), но только конвертирует символы в нижний регистр.

Вызов *widen*(*c*) преобразует символ с в соответствующее значение *Ch*. Если набор символов *Ch* располагает несколькими символами, соответствующими *c*, то согласно стандарту используется «простейшее разумное преобразование». Например:

```
wcout<<use_facet<ctype<wchar_t> >(wcout.getloc()).widen('e');
```

выведет разумный эквивалент символа *e* в контексте локализации потока *wcout*.

С помощью *widen*() можно осуществлять и преобразования между не связанными друг с другом представлениями символов, например ASCII и EBCDIC. Например, предположим, что существует контекст локализации *ebcdic*:

```
char EBCDIC_e = use_facet<ctype<char> >(ebcdic).widen('e');
```

Вызов *widen*(*b*, *e*, *v*) берет каждый символ из интервала [*b*, *e*) и помещает расширенную версию в соответствующую позицию массива *v*.

Вызов *narrow*(*ch*, *def*) выдает значение *char*, соответствующее символу *ch* типа *Ch*. И снова используется «простейшее разумное преобразование». Если не существует *char*, соответствующего *ch*, то возвращается *def*.

Вызов *narrow*(*b*, *e*, *def*, *v*) берет каждый символ из интервала [*b*, *e*) и помещает суженную версию в соответствующую позицию массива *v*.

Общая идея состоит в том, что *narrow*() преобразует из более широкого набора символов в более узкий, а *widen*() — наоборот. Для символа с из более узкого набора мы ожидаем, что

```
c == narrow(widen(c),0)     // не гарантируется
```

Это справедливо при условии, что символ, соответствующий *c*, имеет единственное представление в более узком наборе символов. А это не гарантируется. Если символы, представимые с помощью *char*, не составляют подмножества большего

набора символов типа *Ch*, то следует ожидать проблем в коде, работающем с «символами вообще».

Аналогично, мы могли бы ожидать для символа *ch* из более широкого набора символов, что

```
widen(narrow(ch,def)) == ch || widen(narrow(ch,def)) == widen(def) // не гарантируется
```

Тем не менее, хотя чаще всего это и так, невозможно дать такую гарантию для символов, имеющих несколько значений в расширенном наборе и лишь одно — в более узком наборе символов. Например, число 7 часто имеет несколько представлений в расширенных наборах. Причина здесь в том, что, как правило, большие символьные наборы содержат несколько обычных наборов символов в качестве подмножеств, и символы из меньших наборов реплицируются для удобства преобразования.

Для каждого символа из базового наборам символов (§C.3.3) гарантируется, что

```
widen(narrow(ch_lit,0)) == ch_lit
```

Например:

```
widen(narrow('x',0)) == 'x'
```

Функции ***narrow***() и ***widen***() соблюдают классификацию символов насколько возможно. Например, если справедливо ***is***(***alpha***, ***c***), то также справедливы ***is***(***alpha***, ***narrow***(***c***, '***a***')) и ***is***(***alpha***, ***widen***(***c***)), пока ***alpha*** является корректной маской для текущего контекста локализации.

Основной причиной использования фасета ***ctype*** вообще и функций ***narrow***() и ***widen***() в частности является написание кода, который осуществляет ввод/вывод и манипулирует строками для любого набора символов. Отсюда следует, что реализации ***iostream*** сильнейшим образом зависят от этих средств. Опираясь на <***iostream***> и <***string***>, пользователь может избежать непосредственного использования фасета ***ctype***.

Для фасета ***ctype*** имеется и ***_byname***-версия (§D.4, §D.4.1):

```
template<class Ch> class std::ctype_byname: public ctype<Ch> {/*...*/};
```

D.4.5.1. Вспомогательные шаблоны функций

Наиболее типичным применением фасета ctype является проверка принадлежности символа к некоторой классификации. Для решения этой задачи имеется целый набор шаблонов функций:

```
template<class Ch> bool isspace(Ch c, const locale& loc);
template<class Ch> bool isprint(Ch c, const locale& loc);
template<class Ch> bool iscntrl(Ch c, const locale& loc);
template<class Ch> bool isupper(Ch c, const locale& loc);
template<class Ch> bool islower(Ch c, const locale& loc);
template<class Ch> bool isalpha(Ch c, const locale& loc);
template<class Ch> bool isdigit(Ch c, const locale& loc);
template<class Ch> bool ispunct(Ch c, const locale& loc);
template<class Ch> bool isxdigit(Ch c, const locale& loc);
template<class Ch> bool isalnum(Ch c, const locale& loc);
template<class Ch> bool isgraph(Ch c, const locale& loc);
```

Эти функции тривиально реализуются при помощи фасета ***use_facet***. Например:

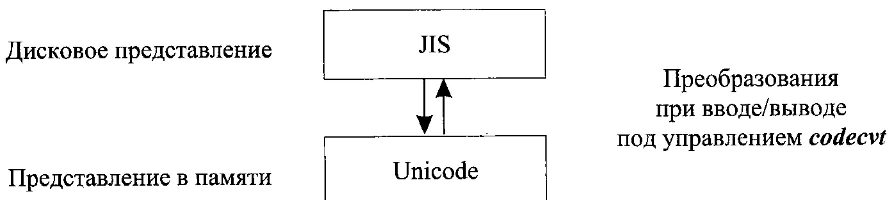
```
template<class Ch>
inline bool isspace(Ch c, const locale& loc)
{
  return use_facet<ctype<Ch> >(loc).is(space,c);
}
```

Версии перечисленных функций с одним аргументом, представленные в §20.4.2, являются теми же функциями для текущего глобального контекста локализации C (но не для ***locale***() языка C++). За исключением редких случаев, когда глобальные контексты локализации C и C++ различаются (§D.2.3), мы можем представлять себе одноаргументные версии как двухаргументные, использующие ***locale***(). Например:

```
inline int isspace(int i)
{
  return isspace(i,locale());    // почти всегда
}
```

D.4.6. Преобразование кодов символов

Иногда представление хранящихся в файле символов отличается от их желаемого представления в оперативной памяти. Например, часто японские символы хранятся в файлах, в которых используются специальные индикаторы для указания, к которому из четырех распространенных символьных наборов *(kanji, katakana, hiragana, romaji)* они принадлежат. Это немного неудобно, поскольку смысл каждого байта зависит от *состояния смещения* (*shift state*). В то же время, это экономит память, поскольку лишь *kanji* требует для представления символов более одного байта. В памяти компьютера символы проще обрабатывать, если они представлены мультибайтными символьными наборами (кодировками), в которых каждому символу соответствует одно и то же количество байт. В таких случаях (например, в случае кодировки Unicode) символы хранятся в памяти в виде расширенных символов (***wchar_t***; §4.3). Вот почему имеется фасет ***codecvt*** для преобразования символов из одной кодировки в другую при их чтении и записи. Например:



Этот механизм преобразования кодировок является достаточно общим для реализации произвольных преобразований представлений символов. Он позволяет писать программы, использующие удобные внутренние представления символов (в виде ***char***, ***wchar_t*** и т.п.), и воспринимающие различные представления символов потока посредством управления контекстом их локализации. Альтернативой была бы необходимость изменения самой программы или необходимость в промежуточных переконвертациях файлов при их чтении и записи.

Фасет ***codecvt*** обеспечивает преобразования между различными наборами символов, когда символ перемещается между буфером потока и внешним хранилищем:

```
class std::codecvt_base
{
public:
  enum result {ok,partial,error,noconv};    // индикаторы результата
};

template<class I, class E, class State>
class std::codecvt: public locale::facet, public codecvt_base
{
public:
  typedef I intern_type;
  typedef E extern_type;
  typedef State state_type;

  explicit codecvt(size_t r = 0);

  result in(State&, const E* from, const E*from_end, const E*& from_next,    // чтение
          I* to, I* to_end, I*& to_next) const;
  result out(State&, const I* from, const I* from_end, const I*& from_next,   // запись
          E* to, E* to_end, E*& to_next) const;

  result unshift(State&, E* to, E* to_end, E*& to_next) const;

  int encoding() const throw();          // свойства перекодирования
  bool always_noconv() const throw();    // можно выполнять ввод/вывод без трансляции?

  int length(const State&, const E* from, const E* from_end, size_t max) const;
  int max_length() const throw();

  static locale::id id;    // объект идентификации фасета (§D.2, §D.3, §D.3.1)

protected:
  ~codecvt();
  // виртуальные "do_"-функции (см. §D.4.1)
};
```

Фасет ***codecvt*** используется ***basic_filebuf*** (§21.5) для чтения или записи символов. Этот фасет ***basic_filebuf*** получает из контекста локализации потока (§21.7.1).

Аргумент шаблона ***State*** является типом, который используется для хранения состояния сдвига подлежащего конвертированию потока. Кроме того, ***State*** можно использовать для идентификации различных преобразований путем уточняющих специализаций. Последний механизм полезен, поскольку символы разнообразных кодировок могут храниться в объектах одного и того же типа. Например:

```
class JISstate {/*..*/};
p = new codecvt<wchar_t, char, mbstate_t>;    // стандартный char в wchar_t
q = new codecvt<wchar_t, char, JISstate>;     // JIS в wchar_t
```

Без аргумента ***State*** не существовало бы способа сообщить фасету, в какой кодировке воспринимать поток символов char. Тип ***mbstate_t*** из <***cwchar***> или <***wchar.h***> идентифицирует стандартное системное преобразование между ***char*** и ***wchar_t***.

Можно также создать новый вариант ***codecvt*** в качестве производного класса. Например:

```
class JIScvt: public codecvt<wchar_t,char,mbstate_t> {/*...*/};
```

Вызов ***in***(***s***, ***from***, ***from_end***, ***from_next***, ***to***, ***to_end***, ***to_next***) читает каждый символ из интервала [***from***, ***from_end***) и пытается преобразовать его. Если символ удается преобразовать, ***in***() записывает его преобразованную форму в соответствующую позицию интервала [***to***, ***to_end***); если не удается — ***in***() завершает на этом свое выполнение. По завершении, ***in***() сохраняет в ***from_next*** позицию за последним прочитанным символом, а в ***to_next*** — позицию за последним записанным символом. Значение ***result***, возвращаемое функцией ***in***(), индицирует выполненный объем работы:

ok: преобразованы все символы из интервала [***from***, ***from_end***).

partial: преобразованы не все символы из интервала [***from***, ***from_end***).

error: функция ***in***() встретила символ, который не смогла преобразовать.

noconv: не потребовалось преобразований.

Заметьте, что возврат ***partial*** не обязательно свидетельствует об ошибке. Возможно, просто нужно прочесть больше символов для завершения многобайтного символа с последующей его записью, или нужно очистить выходной буфер, чтобы в нем оказалось достаточно места для приема символов.

Аргумент ***s*** типа ***State*** индицирует состояние входной последовательности символов в момент вызова функции ***in***(). Это важно для случаев, когда внешнее представление символов использует состояние сдвига (shift state). Обратите внимание на то, что ***s*** является неконстантной ссылкой; под конец вызова ***s*** содержит состояние сдвига входной последовательности. Это позволяет программисту работать с ***partial***-результатом преобразований и конвертировать длинные последовательности посредством нескольких вызовов функции ***in***().

Вызов ***out***(***s***, ***from***, ***from_end***, ***from_next***, ***to***, ***to_end***, ***to_next***) преобразует [***from***, ***from_end***) из внутреннего представление во внешнее так же, как ***in***() преобразует из внешнего представления во внутреннее.

Поток символов должен начинаться и заканчиваться в «нейтральном» состоянии (без сдвига — unshifted state). Как правило, это состояние ***State***(). Вызов ***unshift***(***s***, ***to***, ***to_end***, ***to_next***) анализирует ***s*** и помещает символы в [***to***, ***to_end***) таким образом, что последовательность символов возвращается в нейтральное состояние. Возврат ***unshift***() и использование аргумента ***to_next*** вполне аналогичны таковым для функции ***out***().

Вызов ***length***(***s***, ***from***, ***from_end***, ***max***) возвращает число символов из [***from***, ***from_end***), которое функция ***in***() может преобразовать.

Функция ***encoding***() возвращает:

- ***-1*** — если внешнее представление символов использует состояние (есть сдвиг, нет сдвига)
- ***0*** — если кодировка символов использует переменное число байт (например, отдельный бит может сообщать о том, один или два байта отводятся под символ)
- ***n*** — если каждый символ во внешнем представлении занимает ***n*** байт

Вызов ***always_noconv***() возвращает ***true***, если не требуется преобразований между внутренним и внешним представлениями символов, и возвращает ***false*** в противном случае. Ясно, что ***always_noconv***()==***true*** открывает возможность максимально эффективной реализации, которая просто не вызывает функции преобразования.

Вызов ***max_length***() возвращает максимальное значение, которое может вернуть ***length***() при корректных аргументах.

Простейшее преобразование кодировок, которое я смог придумать, это преобразование входных символов в верхний регистр. Так что ниже представлен возможно простейший вариант ***codecvt***, который выполняет все-таки некоторую работу:

```
class Cvt_to_upper : public codecvt<char, char, mbstate_t>
{
  explicit Cvt_to_upper(size_t r = 0) : codecvt(r) { }

protected:
  // читает внешнее представление - пишет внутреннее:
  result do_in(State& s, const char* from, const char* from_end, const char*&
              from_next, char* to, char* to_end, char*& to_next) const;

  // читает внутреннее представление - пишет внешнее:
  result do_out(State& s, const char* from, const char* from_end, const char*&
               from_next, char* to, char* to_end, char*& to_next) const
  {
    return codecvt<char, char, mbstate_t>::do_out
         (s, from, from_end, from_next, to, to_end, to_next);
  }

  result do_unshift(State&, E* to, E* to_end, E*& to_next) const { return ok; }

  int do_encoding() const throw() { return 1; }
  bool do_always_noconv() const throw() { return false; }

  int do_length(const State&, const E*from, const E* from_end, size_t max) const;
  int do_max_length() const throw();
};

codecvt<char, char, mbstate_t>::result
Cvt_to_upper::do_in(State& s, const char* from, const char* from_end,
              const char*& from_next, char* to, char* to_end,
              char*& to_next) const
{
  // ... §D.6[16] ...
}

int main()                // тривиальный тест
{
  locale ulocale(locale(), new Cvt_to_upper);

  cin.imbue(ulocale);

  char ch;
  while(cin>>ch) cout << ch;
}
```

Для ***codecvt*** имеется и ***_byname***-версия (§D.4, §D.4.1):

```
template<class I, class E, class State>
class std::codecvt_byname : public codecvt<I, E, State> { /*...*/ };
```

D.4.7. Сообщения

Неудивительно, что большинство пользователей предпочитает взаимодействовать с программой на своем родном языке. Однако мы не можем обеспечить универсальный стандартный механизм взаимодействия с программой в точном соответствии с контекстом локализации. Вместо этого, стандартная библиотека предоставляет нехитрый механизм хранения зависящих от локализации строк, из которых программист может составлять простые сообщения. По существу, класс ***messages*** (сообщения) реализует тривиальную базу данных, доступную только для чтения:

```
class std::messages_base
{
public:
  typedef int catalog;        // тип идентификатора каталога
};

template<class Ch>
class std::messages: public locale::facet, public messages_base
{
public:
  typedef Ch char_type;
  typedef basic_string<Ch> string_type;

  explicit messages (size_t r = 0) ;

  catalog open (const basic_string<char>& fn, const locale&) const;
  string_type get (catalog c, int set, int msgid, const string_type& d) const;
  void close (catalog c) const;

  static locale::id id;       // объект идентификации фасета (§D.2, §D.3, §D.3.1)

protected:
  ~messages () ;
  // виртуальные "do_"-функции (см. §D.4.1)
};
```

Вызов ***open***(***s***, ***loc***) открывает «каталог» сообщений ***s*** для контекста локализации ***loc***. Каталог — это набор строк, организованных зависящим от реализации способом и доступных посредством функции ***messages***::***get***(). Если открыть каталог с именем ***s*** не удается, то возвращается отрицательное значение. Каталог должен быть открыт до первого вызова функции ***get***().

Вызов ***close***(***cat***) закрывает каталог, идентифицируемый посредством ***cat***, и освобождает все связанные с ним ресурсы.

Вызов ***get***(***cat***, ***set***, ***id***, ***"foo"***) ищет в каталоге cat сообщение, идентифицируемое посредством (***set***, ***id***). Если строка находится, то ***get***() возвращает эту строку; в противном случае ***get***() возвращает строку по умолчанию (здесь это ***string***(***"foo"***)).

Приведем пример фасета сообщений для системы, в которой каталог сообщений реализован в виде вектора множеств «сообщений», имеющих тип ***string***:

```
struct Set
{
  vector<string> msgs;
};
```

```
struct Cat
{
  vector<Set> sets;
};

class My_messages: public messages<char>
{
  vector<Cat>& catalogs;

public:
  explicit My_messages(size_t = 0) : catalogs(*new vector<Cat>) { }

  catalog do_open(const string& s, const locale& loc) const;    // открыть каталог s
  string do_get(catalog c, int s, int m, const string&) const; // взять сообщ-е (s,m) из c
  void do_close(catalog cat) const
  {
    if(catalogs.size() <= cat) catalogs.erase(catalogs.begin()+cat);
  }

  ~My_messages() {delete &catalogs;}
};
```

Все функции-члены объявлены константными, потому что структура данных каталога (**vector<Set>**) хранится вне фасета.

Сообщения выбираются указанием каталога, множества внутри каталога и строки внутри множества. Строка-аргумент используется в качестве возврата по умолчанию в случае, если сообщение в каталоге не обнаруживается.

```
string My_messages::do_get(catalog cat, int set, int msg, const string& def) const
{
  if(catalogs.size() <= cat) return def;
  Cat& c = catalogs[cat];
  if(c.sets.size() <= set) return def;
  Set& s = c.sets[set];
  if(s.msgs.size() <= msg) return def;
  return s.msgs[msg];
}
```

Открытие каталога включает в себя считывание текстового содержимого с диска в структуру ***Cat***. В примере я выбрал представление, читать которое не составляет никакого труда. Множество разграничивается при помощи <<< и >>>, и каждое сообщение есть одна строка текста:

```
messages<char>::catalog My_messages::do_open(const string& n, const locale& loc)
const
{
  string nn = n + locale().name();
  ifstream f(nn.c_str());
  if(!f) return -1;

  catalogs.push_back(Cat());
  Cat& c = catalogs.back();
  string s;
  while (f>>s && s=="<<<")                                     // читаем Set
  {
```

```
    c.sets.push_back(Set());
    Set& ss = c.sets.back();
    while (getline(f,s) && s != ">>>") ss.msgs.push_back(s);    // читаем сообщение
  }
  return catalogs.size() -1;
}
```

Вот пример простейшего использования:

```
int main()
{
  if(!has_facet<My_messages>(locale()))
  {
    cerr << "no messages facet found in " << locale().name() << '\n';
    exit(1);
  }

  const messages<char>& m = use_facet<My_messages>(locale());
  extern string message_directory;                    // здесь я храню сообщения
  int cat = m.open(message_directory, locale());

  if(cat<0)
  {
    cerr << "no catalog found\n";
    exit(1);
  }

  cout << m.get(cat,0,0, "Missed again!") << endl;
  cout << m.get(cat,1,2, "Missed again!") << endl;
  cout << m.get(cat,1,3, "Missed again!") << endl;
  cout << m.get(cat,3,0, "Missed again!") << endl;
}
```

Если каталог содержит

```
<<<
hello
goodbye
>>>
yes
no
maybe
>>>
```

то программа выведет

```
hello
maybe
Missed again!
Missed again!
```

D.4.7.1. Использование сообщений из других фасетов

Помимо того, что сообщения используются в качестве хранилища зависящих от локализации строк, которые используются для общения с пользователем, они могут

содержать строки для других фасетов. Например, фасет ***Season_io*** (§D.3.2) можно написать следующим образом:

```
class Season_io : public locale : :facet
{
  const messages<char>& m;                  // директория сообщений
  int cat;                                  // каталог сообщений

public:
  class Missing_messages { };

  Season_io (int i = 0)  : locale : :facet (i) ,
m (use_facet<Season_messages> (locale ( ) ) ) ,
                        cat (m . open { message_directory, locale ( ) ) )
  {
    if (cat<0) throw Missing_messages ( ) ;
  }

  ~Season_io ( )  { }        // чтобы можно было уничтожать объекты Season_io (§D.3)

  const string& to_str (Season x) const;            // строковое представление x
  bool from_str (const string& s, Season& x) const; // поместить Season соответ-ий s в x

  static locale : :id id;      // объект идентификации фасета (§D.2. §D.3, §D.3.1)
};

locale : :id Season_io : :id;   // определяем идентифицирующий объект

const string& Season_io : :to_str (Season x) const
{
  return m->get (cat,x, "no-such-season") ;
}

bool Season_io : :from_str (const string& s, Season& x) const
{
  for (int i = Season : :spring; i<=Season : :winter; i++)
    if (m->get (cat, i, "no-such-season") == s)
    {
      x = Season (i) ;
      return true;
    }
  return false;
}
```

Это решение, использующее механизм ***messages***, отличается от исходного (§D.3.2) тем, что разработчик строк ***Season*** для нового контекста локализации должен помещать их в каталог ***messages***. Это не может представлять сложности для тех, кто в состоянии добавлять в систему новые контексты локализации. Однако из-за того, что ***messages*** предоставляют механизм доступа «только для чтения», добавление нового набора названий времен года может оказаться не по плечу прикладному программисту.

Имеется и версия messages с суффиксом ***_byname*** (§D.4, §D.4.1):

```
template<class Ch>
class std : :messages_byname: public messages<Ch> { /*...*/ };
```

D.5. Советы

1. Будьте готовы к тому, что любая нетривиальная программа или система, непосредственно взаимодействующие с людьми, будут эксплуатироваться в разных странах; §D.1.
2. Не рассчитывайте на то, что все используют тот же набор символов, что и вы; §D.4.1.
3. Используйте *locale*, а не частные проприетарные решения для ввода/вывода, зависящего от национальных и культурных особенностей; §D.1.
4. Не помещайте строковые имена локализаций непосредственно в текст программы; §D.2.1.
5. Минимизируйте применение глобальной форматной информации; §D.2.3, §D.4.4.7.
6. Предпочитайте зависящие от локализации сравнения строк и сортировки; §D.2.4, §D.4.1.
7. Создавайте фасеты неизменяемыми (immutable); §D.2.2, §D.3.
8. Сконцентрируйте модификацию контекстов локализации в четко ограниченных фрагментах программы; §D.2.3.
9. Пусть время жизни фасетов регулируется из контекстов локализации; §D.3.
10. При написании функций ввода/вывода, зависящих от локализации, не забывайте обрабатывать исключения, генерируемые пользовательскими (замещенными) функциями; §D.4.2.2.
11. Используйте простой тип *Money* для хранения денежных сумм; §D.4.3.
12. Используйте простые пользовательские типы для хранения значений, требующих ввода/вывода, чувствительного к местным и национальным особенностям (а не прибегайте к приведению типов ко встроенным типам до и после операций чтения/записи); §D.4.3.
13. Не доверяйте измерениям времени выполнения до тех пор, пока вы не учтете все влияющие на них факторы; §D.4.4.1.
14. Не забывайте об ограничениях типа *time_t*; §D.4.4.1, §D.4.4.5.
15. Используйте процедуры ввода даты, допускающие различные форматы дат; §D.4.4.5.
16. Отдавайте предпочтение тем функциям классификации символов, в которых локализации учитываются (используются) явным образом; §D.4.5, §D.4.5.1.

D.6. Упражнения

1. (*2.5) Определите *Season_io* (§D.3.2) для языка, отличного от американского английского.
2. (*2) Определите класс *Season_io* (§D.3.2), который принимает строки названий времен года в качестве аргументов конструктора, с тем, чтобы названия времен года в разных локализациях могли представляться объектами этого класса.

3. (*3) Напишите функцию ***collate<char>::compare()***, которая работает согласно словарному упорядочению. Сделайте это для языков, вроде французского или немецкого, в которых букв больше, чем в английском.
4. (*2) Напишите программу, которая читает и пишет логические значения (типа ***bool***) в виде чисел, английских слов или слов любого другого языка по вашему выбору.
5. (*2.5) Определите тип ***Time*** для представления времени суток. Определите тип ***Date_and_time*** с помощью типов ***Time*** и ***Date***. Обсудите плюсы и минусы такого подхода в сравнении с ***Date*** из §D.4.4. Реализуйте чувствительный к локализации ввод/вывод для ***Time*** и ***Date_and_time***.
6. (*2.5) Разработайте и реализуйте фасет почтового индекса. Сделайте это как минимум для двух стран с непохожими соглашениями относительно написания адресов. Например, ***NJ 07932*** и ***CB21QA***.
7. (*2.5) Разработайте и реализуйте фасет телефонных номеров. Сделайте это как минимум для двух стран с непохожими соглашениями относительно написания телефонных номеров. Например, ***(973) 360-8000*** и ***1223 343000***.
8. (*2.5) Поэкспериментируйте и определите, какие форматы ввода/вывода использует ваша система для дат.
9. (*2.5) Определите функцию ***get_time()***, которая «угадывает» смысл неоднозначных представлений для дат, таких как 12/5/1995, но по-прежнему отвергает все или почти все ошибочные записи. Решите, какие именно догадки приемлемы и какова вероятность ложных срабатываний.
10. (*2) Определите ***get_time()***, которая воспринимает гораздо больше входных форматов, чем вариант из §D.4.4.5.
11. (*2) Составьте список контекстов локализации, поддерживаемых вашей системой.
12. (*2.5) Попробуйте выяснить, где в вашей системе хранятся именованные локализации. Если вы имеете доступ к той части системы, где хранятся контексты локализации, создайте и сохраните новую именованную локализацию. Постарайтесь не испортить существующие локализации.
13. (*2) Сравните две реализации ***Season_io*** (§D.3.2 и §D.4.7.1).
14. (*2) Напишите и протестируйте фасет ***Date_out***, который записывает даты типа ***Date*** с помощью формата, передаваемого в качестве аргумента конструктору. Обсудите плюсы и минусы такого подхода в сравнении с глобальным форматом дат, предоставляемым ***date_fmt*** (§D.4.4.6).
15. (*2.5) Реализуйте ввод/вывод римских чисел (вида ***XI*** или ***MDCLII***).
16. (*2.5) Реализуйте и протестируйте ***Cvt_to_upper*** (§D.4.6).
17. (*2.5) Используйте ***clock()*** для определения стоимости (1) функционального вызова, (2) вызова виртуальной функции, (3) чтения ***char***, (4) чтения int из одной цифры, (5) чтения ***int*** из 5 цифр, (6) чтения ***double*** из 5 цифр, (7) односимвольного ***string***, (8) пятисимвольного ***string*** и (9) сорокасимвольного ***string***.
18. (*6.5) Выучите новый иностранный язык.

Приложение E

Исключения и безопасность стандартной библиотеки

Все будет работать так, как вы ожидаете, лишь в случае,
когда ваши ожидания корректны.
— Хайман Роузен

Исключения и безопасность — технологии реализации безопасности при исключениях — явное управление памятью — присваивание — метод ***push_back***() — конструкторы и инварианты — гарантии стандартных контейнеров — вставка и удаление элементов — гарантии и компромиссы — функция ***swap***() — инициализация и итераторы — ссылки на элементы — предикаты — строки, потоки, алгоритмы — типы ***valarray*** и ***complex*** — стандартная библиотека С — рекомендации пользователям стандартной библиотеки — советы — упражнения.

E.1. Введение

Функции стандартной библиотеки часто используют операции, предоставляемые пользователем в виде аргументов функции или шаблона. Очевидно, что некоторые из этих операций могут иногда генерировать исключения. Также генерировать исключения могут и функции выделения динамической памяти. Рассмотрим пример:

```
void f(vector<X>& v, const X& g)
{
  v[2] = g;            // X-ое присваивание может генерировать исключение
  v.push_back(g);  // аллокатор у vector<X> может генерировать исключение
  sort(v.begin(), v.end());        // X-ая операция < может генерировать исключение
  vector<X> u = v;  // X-ый копирующий конструктор может генерировать исключение
  // ...
  // и здесь уничтожается - нужно гарантировать, что X-ый деструктор сработает
  // корректно
}
```

Что произойдет, если при выполнении присваивания значения *g* будет сгенерировано исключение? Будет ли при этом вектор v оставлен в недействительном состоянии? Что случится, если конструктор, вызываемый в процессе копирования *g* операцией *v.push_back*(), сгенерирует исключение *std*::*bad_alloc*? Изменится ли при этом число элементов? Или в контейнер (вектор) добавится недействительный элемент? Что произойдет, если операция «меньше чем» из типа *X* сгенерирует исключение во время сортировки? Элементы будут отсортированы частично? Или операция сортировки удалит элемент из контейнера и не вернет его назад?

Формирование окончательного списка возможных исключений в рассмотренном примере оставлено в качестве самостоятельного упражнения (§Е.8[1]). Объяснение того, как может данный пример корректно работать при любом хорошо определённом типе *X* — даже если этот тип генерирует исключения — является одной из задач данного приложения. Львиная доля приложения сводится при этом к объяснению, что такое «корректно работать» и что такое «хорошо определенный тип» в контексте исключений, а также к выработке соответствующей терминологии.

Перечислим цели данного приложения:

1. Рассказать, как пользователь должен проектировать типы, удовлетворяющие требованиям стандартной библиотеки.
2. Сформулировать гарантии, предоставляемые стандартной библиотекой.
3. Сформулировать требования стандартной библиотеки к коду, предоставляемому пользователем.
4. Продемонстрировать эффективные технологии построения эффективных контейнеров, безопасных в контексте исключений.
5. Представить несколько правил программирования безопасного в контексте исключений кода.

Обсуждение проблемы *безопасности в контексте исключений* (*exception safety*) неизбежно сосредотачивается на самых худших случаях. В каких случаях исключения доставляют наибольшие проблемы? Как стандартная библиотека защищает себя и пользователей библиотеки от возникающих при этом неприятностей? Как сами пользователи могут помочь в преодолении проблем? Несмотря на все эти обсуждения, не дайте отвлечь себя от центральной идеи о том, что исключения, тем не менее, являются наилучшим способом сообщений о возникновении ошибочных ситуаций в работающих программах (§14.1, §14.9).

Обсуждение концепций, технологий и гарантий стандартной библиотеки организовано следующим образом:

- §Е.2 Содержит обсуждение проблемы безопасности в контексте исключений.
- §Е.3 Рассматривает методы реализации контейнеров и операций, безопасных в контексте исключений.
- §Е.4 Уточняет гарантии, которые предоставляет стандартная библиотека в отношении контейнеров и операций.
- §Е.5 Суммирует вопросы, связанные с безопасностью в контексте исключений, для остальных частей стандартной библиотеки.
- §Е.6 Представляет обзор вопросов безопасности в контексте исключений с точки зрения пользователя стандартной библиотеки.

Стандартная библиотека предоставляет наглядные примеры вопросов безопасности, которые нужно принимать к сведению при разработке ответственных приложений. Методы, которые используются в стандартной библиотеке для обеспечения безопасности в контексте исключений, применимы к широкому диапазону проблем.

Е.2. Исключения и безопасность

Говорят, что операция, выполняемая на объекте, *безопасна в контексте исключений* (*exception safe*), если операция оставляет объект в *действительном состоянии* (*valid state*) при генерации исключения. Действительное состояние может быть и ошибочным состоянием, которое для восстановления требует коррекции (очистки — cleanup), но оно обязано быть полностью известным (хорошо определенным — well defined). Именно последнее обстоятельство позволяет написать код для обработки ошибочного состояния объекта. Этот код обработки исключения может, например, уничтожить объект, восстановить его исходное состояние, повторить операцию, просто продолжить работу программы и т.д.

Иными словами, у объекта есть инвариант (§24.3.7.1), конструкторы устанавливают этот инвариант, дальнейшие операции соблюдают этот инвариант (даже при генерации исключений), а деструктор выполняет заключительную «очистку». Операции должны заботиться о том, чтобы инвариант соблюдался перед генерацией исключений с тем, чтобы объект находился в действительном состоянии. Вполне вероятно, что это действительное состояние может не устраивать приложение. Например, строка может быть пустой, или контейнер может оставаться в неотсортированном состоянии. Таким образом, восстановить объект — значит придать ему значение, устраивающее приложение больше, чем значение, которое объект имел перед началом выполнения неудавшейся операции. В этом отношении для нас самыми интересными объектами стандартной библиотеки являются контейнеры.

Далее мы рассмотрим, при каких условиях операции на контейнерах стандартной библиотеки могут считаться безопасными в контексте возможных исключений. Всего возможны две концептуально чистые стратегии:

1. «*Без гарантий*» — если сгенерировалось исключение, обрабатываемый контейнер возможно остался в испорченном состоянии.
2. «*Сильная гарантия*» — если сгенерировалось исключение, то обрабатываемый контейнер остался в состоянии, которое он имел перед началом выполнения операции стандартной библиотеки.

К сожалению, оба случая неприемлемы для реального использования. Случай [1] неприемлем, потому что в этом случае после возникновения исключения в контейнерной операции к контейнеру уже нельзя обратиться — его даже уничтожить нельзя без опасений спровоцировать ошибку времени выполнения. Случай [2] неприемлем, поскольку он навязывает каждой операции стандартной библиотеки слишком дорогую *семантику отката* (*roll-back semantics*).

Чтобы разрешить эту дилемму, стандартная библиотека формулирует такой набор гарантий безопасности исключений, который позволяет разделить ответственность за создание надежно работающих программ между разработчиками стандартной библиотеки и ее пользователями:

3a. «*Базовая гарантия* (*basic guarantee*) *для всех операций*» — поддерживаются базовые инварианты стандартной библиотеки и нет утечек ресурсов, например, памяти.

3b. «*Сильная гарантия* (*strong guarantee*) *для ключевых операций*» — в дополнение к базовой гарантии операции либо выполняются успешно, либо не приводят ни к какому эффекту. Эта гарантия обеспечивается для таких ключевых операций, как ***push_back***(), одноэлементных ***insert***() для списка ***list*** и ***uninitialized_copy***() (§E.3.1, §E.4.1).

3c. «*Гарантия отсутствия исключений* (*nothrow guarantee*) *для некоторых операций*» — дополнительно к базовой гарантии устанавливается, что некоторые операции не генерируют исключений. Эта гарантия дается для небольшого числа простых операций типа ***swap***() или ***pop_back***() (§E.4.1).

Как базовая, так и сильная гарантии обеспечиваются при условии, что предоставляемые пользователем операции (такие как операции присваивания и функции ***swap***()) не оставляют элементы контейнера в недействительных состояниях, не допускают утечек ресурсов и что деструкторы не генерируют исключений. Например, рассмотрим два класса, имеющих характер дескрипторных классов (§25.7):

```
template<class T> class Safe
{
  T* p;
public:
  Safe() : p(new T) { }
  ~Safe() {delete p; }
  Safe& operator= (const Safe& a) { *p = *a.p; return *this; }
  // ...
};

template<class T> class Unsafe
{
  T* p;
public:
  Unsafe(T* pp) : p(pp) { }
  ~Unsafe() {if(!p->destructible()) throw E(); delete p; }

  Unsafe& operator= (const Unsafe& a)
  {
    p->~T();                    // уничтожаем старое значение (§10.4.11)
    new (p) T(*a.p);            // конструируем копию *a.p в *p (§10.4.11)
    return *this;
  }
  // ...
};

void f(vector<Safe<Some_type> >& vg, vector<Unsafe<Some_type> >& vb)
{
  vg.at(1) = Safe<Some_type>();
  vb.at(1) = Unsafe<Some_type>(new Some_type);
  // ...
}
```

В этом примере создание ***Safe*** выполняется успешно, только если успешно создается ***T***. Создание ***T*** может не состояться из-за неудачи с выделением памяти (с генерацией ***std::bad_alloc***) или в случае генерации исключения конструктором ***T***. Но в каждом удачно созданном ***Safe*** поле ***p*** указывает на успешно созданный ***T***; если же конструктор потерпит неудачу, то ***T*** (а равно и ***Safe***) не создается. Точно так же операция присваивания ***T*** может сгенерировать исключение, заставляя операцию присваивания ***Safe*** повторно сгенерировать это исключение. Проблем не возникает, если операция присваивания у типа ***T*** всегда оставляет свои операнды в действительном состоянии. Поэтому ***Safe*** — корректно ведущий себя класс и, следовательно, каждая операция стандартной библиотеки над ***Safe*** будет иметь разумный и хорошо определенный результат.

С другой стороны, ***Unsafe*** написан небрежно (точнее, написан тщательно с целью продемонстрировать, как поступать не нужно). Конструирование ***Unsafe*** никогда не потерпит неудачи, а операции с ***Unsafe***, вроде операций присваивания и уничтожения, оставлены наедине со множеством потенциальных проблем. Операция присваивания может потерпеть неудачу при генерации исключения в копирующем конструкторе ***T***. Это оставит ***T*** в неопределенном состоянии, поскольку старое значение ****p*** было уничтожено, а новое значение его не заменило. В общем случае результат непредсказуем. Деструктор ***Unsafe*** демонстрирует непродуманную попытку защититься от нежелательного уничтожения объекта: генерация исключения в процессе обработки исключения приведет к вызову ***terminate()*** (§14.7), в то время как стандартная библиотека требует нормального возврата из деструктора после уничтожения объекта. Стандартная библиотека не дает — и не может дать — никаких гарантий, если пользователь предоставляет объекты, которые ведут себя столь плохо.

С точки зрения обработки исключений, типы ***Safe*** и ***Unsafe*** различаются тем, что ***Safe*** использует свой конструктор для установки инварианта (§24.3.7.1), позволяющего реализовать операции просто и эффективно. Если установить инвариант не удается, то генерируется исключение до того, как будет сконструирован недействительный объект. А тип ***Unsafe*** действует наобум без какого-либо значимого инварианта, его операции генерируют исключения без привязки к какой-либо единой стратегии обработки ошибок. Естественно, это приводит к нарушению предположений (разумных) стандартной библиотеки относительно поведения типов. Например, ***Unsafe*** может оставить в контейнере недействительные элементы после генерации исключения из ***T::operator=()***, а также может сгенерировать исключение в своем деструкторе.

Отметим, что гарантии стандартной библиотеки касательно предоставляемых пользователем неблагополучных операций аналогичны гарантиям языка касательно нарушений базовой системы типов. Если базовая операция применяется не в соответствии с ее спецификациями, то результирующее поведение не определено. Например, если в деструкторе элемента контейнера vector генерируется исключение, то имеется не больше оснований надеяться на разумный результат, чем в случае разыменования указателя, инициализированного случайным числом:

```
class Bomb
{
public:
  // ...
  ~Bomb() {throw Trouble();}
};
```

```
vector<Bomb> b(10);                          // приводит к неопределенному поведению

void f()
{
  int* p = reinterpret_cast<int*>(rand());   // приводит к неопределенному поведению
  *p = 7;
}
```

Можно твердо заявить, что если вы соблюдаете основные правила языка и стандартной библиотеки, то библиотечные средства будут вести себя корректно, даже когда вы генерируете исключения.

Кроме достижения безопасности исключений как таковой еще стремятся избежать и утечек ресурсов, то есть операция, генерирующая исключение, не только должна оставить свои операнды в действительном состоянии, но и должна обеспечить (в конечном счете) освобождение захваченных ею ресурсов. Например, в точке генерации исключения выделенная ранее память должна либо освобождаться, либо находиться под управлением объекта, который должен нести ответственность за ее дальнейшее освобождение.

Стандартная библиотека гарантирует отсутствие утечек ресурсов при условии, что предоставляемые пользователем операции, вызываемые библиотекой, сами не порождают никаких утечек. Рассмотрим пример:

```
void leak(bool abort)
{
  vector<int> v(10);                                  // нет утечек
  vector<int>* p = new vector<int>(10);               // возможны утечки памяти
  auto_ptr<vector<int> > q(new vector<int>(10));      // нет утечек (§14.4.2)

  if(abort) throw Up();
  // ...
  delete p;
}
```

Здесь при генерации исключения вектор ***v*** и вектор, находящийся под управлением ***q***, будут корректно уничтожены так, что их ресурсы освободятся. В то же время, вектор, на который указывает ***p***, не защищен от исключений и не будет уничтожен. Чтобы сделать эту часть кода безопасной, мы должны либо явным образом вызвать операцию ***delete*** для указателя ***p*** перед генерацией исключения, либо гарантировать, что вектор находится под управлением объекта — такого как ***auto_ptr*** (§14.4.2) — который сам позаботится о надлежащем уничтожении вектора при возникновении исключения.

Отметим, что языковые правила создания и освобождения объектов по частям гарантируют, что исключения, генерируемые в момент создания подобъектов или членов, будут корректно обработаны без специальных мер со стороны кода стандартной библиотеки (§14.4.1). Это правило лежит в основе всех технологий обработки исключений.

Помните также, что память не является единственным ресурсом, подверженным утечке. Открытые файлы, блокировки, сетевые соединения и *потоки управления (threads)* — вот примеры системных ресурсов, которые функция перед генерацией исключения должна либо освободить, либо передать некоторому объекту для последующего их освобождения.

E.3. Технологии реализации безопасности при исключениях

Как всегда, стандартная библиотека демонстрирует примеры проблем, которые могут встречаться во многих других контекстах, и широко применимые методы их преодоления. Для написания безопасного кода имеются следующие базисные приемы и методы:

1. *try*-блок (§8.3.1).
2. Технология «выделение ресурса есть инициализация» (§14.4)
3. Не терять информацию, пока ей не найдена замена.
4. Оставлять объекты в действительном состоянии при генерации (или повторной генерации) исключений.

Действуя указанным образом, мы всегда сможем восстановиться после возникновения ошибочных ситуаций. Практическая трудность в следовании данным рекомендациям заключается в том, что невинно выглядящие операции (вроде <, = и *sort*()) могут генерировать исключения. Требуется опыт, чтобы точно знать, что можно ожидать от приложения.

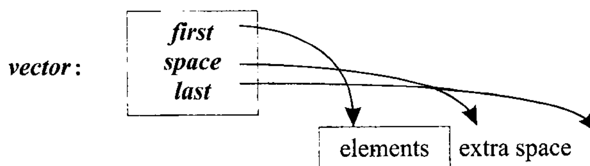
Если вы пишите библиотеку, то в идеале нужно стремиться к сильной гарантии безопасности исключений (§E.2), безусловно реализуя базовую гарантию. При написании обычных программ о безопасности исключений обычно заботятся в меньшей степени. Например, если я пишу простую программу анализа данных для личного пользования, меня вполне устроит ее аварийное завершение в маловероятном случае исчерпания виртуальной памяти. Однако отметим, что безопасность программы при возникновении исключений и общая ее корректность тесно связаны между собой.

Технологии и методы обеспечения безопасности при исключениях, такие как определение и проверка инвариантов (§24.3.7.1), аналогичны методам, которые делают программу компактной и корректной. Отсюда вытекает, что накладные расходы на обеспечение безопасности исключений (базовая гарантия; §E.2), даже в случае сильной гарантии, могут оказаться минимальными или даже совсем пренебрежимыми; см. §E.8[17].

Ниже мы рассмотрим реализацию стандартного контейнера vector (§16.3) и выясним, что требуется для достижения этого идеала, и где можно обойтись менее жесткими требованиями к безопасности.

E.3.1. Простой вектор

Типичная реализация типа vector использует представление, содержащее указатели на первый элемент, на элемент, следующий за последним элементом, и на область памяти за выделенным под элементы блоком памяти (§17.1.3) (возможно эквивалентное представление с указателем и смещениями):



Вот упрощенное объявление типа ***vector***, призванное показать лишь то, что имеет отношение к обсуждению безопасности при исключениях и к предотвращению утечек ресурсов:

```
template<class T, class A = allocator<T> >
class vector
{
private:
  T* v;          // начало выделенной памяти
  T* space;      // конец последовательности эл-ов (начало свобод. прост-ва)
  T* last;       // конец выделенной памяти
  A alloc;       // аллокатор

public:
  explicit vector(size_type n, const T& val = T(), const A& = A());

  vector(const vector& a);               // копирующий конструктор
  vector& operator=(const vector& a);    // присваивание

  ~vector();

  size_type size() const {return space-v;}
  size_type capacity() const {return last-v;}

  void push_back(const T&);
  //...
};
```

Сначала рассмотрим бесхитростную реализацию конструктора:

```
template<class T, class A>
vector<T,A>::vector(size_type n, const T& val, const A& a)
          : alloc(a)                        // копируем аллокатор
{
  v = alloc.allocate(n);                    // выделяем память под эл-ты (§19.4.1)
  space = last = v+n;
  for(T* p=v; p!=last; ++p) a.construct(p,val); // создаем копию val в *p (§19.4.1)
}
```

Здесь имеются три источника исключений:

1. Функция ***allocate***() генерирует исключение, индицирующее отсутствие доступной памяти.
2. Копирующий конструктор аллокатора генерирует исключение.
3. Копирующий конструктор элемента типа ***T*** генерирует исключение, если не может скопировать ***val***.

Во всех случаях никакого объекта не создается, так что деструктор класса vector не вызывается (§14.4.1).

Когда функция ***allocate***() терпит неудачу, она завершает работу по оператору ***throw*** прежде, чем будут выделены какие-либо ресурсы, так что все в порядке.

Когда копирующий конструктор типа ***T*** терпит неудачу, мы уже располагаем выделенной памятью, которую нужно во избежание утечек освободить. Более серьезная проблема состоит в том, что этот копирующий конструктор может сгенерировать исключение после корректного создания некоторого количества элементов, но не всех элементов.

Чтобы справиться с проблемой в этом случае, мы могли бы отслеживать, какие элементы были созданы, и уничтожить их (и только их) в случае ошибки:

```
template<class T, class A>
vector<T,A>::vector(size_type n, const T& val, const A& a)
        : alloc(a)                              // копируем аллокатор
{
  v = alloc.allocate(n);                        // выделяем память под эл-ты
  iterator p;

  try
  {
    iterator end = v+n;
    for(p=v; p!=end; ++p) alloc.construct(p,val); // конструируем элемент (§19.4.1)
    last = space = p;
  }
  catch(...)
  {
    for(iterator q = v; q!=p; ++q) alloc.destroy(q); // уничтожаем созданные элементы
    alloc.deallocate(v,n);                      // освобождаем память
    throw;                                      // повторно генерируем исключение
  }
}
```

Здесь накладные расходы связаны лишь с *try*-блоком. В хорошей реализации C++ эти накладные расходы пренебрежимо малы по сравнению со стоимостью выделения памяти и инициализации элементов. Для реализаций, где организация *try*-блока накладна, заслуживает внимание проверка *if(n)* перед входом в *try*-блок и отдельная обработка случая пустого вектора.

Основную работу для этого конструктора выполняет реализация *uninitialized_fill()*, безопасная в контексте исключений:

```
template<class For, class T>
void uninitialized_fill(For beg, For end, const T& x)
{
  For p;
  try
  {
    for(p=beg; p!=end; ++p)
      new (static_cast<void*>(&*p)) T(x);       // создаем копию x в *p (§10.4.11)
  }
  catch(...)
  {
    for(For q = beg; q!=p; ++q) (&*q)->~T();    // §10.4.11
    throw;
  }
}
```

Любопытная конструкция `&*p` «заботится» об итераторах, которые не являются указателями. В этом случае мы берем адрес разыменованного элемента с целью получения указателя. Явное приведение к типу *void** гарантирует, что используется стандартная библиотечная размещающая функция (§19.4.5), а не какая-либо поль-

зовательская функция operator *new*() для *T**. Этот код оперирует на довольно низком уровне, где написание по настоящему общего кода затруднено.

По счастью, нам нет нужды повторно реализовывать ***uninitialized_fill***(), потому что стандартная библиотека обеспечивает для нее желаемую сильную гарантию (§E.2). Часто важно, чтобы инициализирующая операция либо завершалась успешно, проинициализировав все элементы, либо она завершалась неудачей, не оставляя за собой никаких созданных элементов. Поэтому алгоритмы ***uninitialized_fill***(), ***uninitialized_fill_n***() и ***uninitialized_copy***() (§19.4.4) стандартной библиотеки обеспечивают такую сильную гарантию безопасности исключений (§E.4.4).

Обратите внимание на то, что алгоритм ***uninitialized_fill***() не защищает от исключений, которые генерируются деструкторами элементов или итераторными операциями (§E.4.4). Такая защита была бы слишком дорогой (см. §E.8[16-17]).

Алгоритм ***uninitialized_fill***() может применяться ко множеству видов последовательностей. Соответственно, он использует прямой (однонаправленный) итератор (§19.2.1) и не может гарантировать уничтожения элементов в порядке, обратном их созданию.

Используя ***uninitialized_fill***(), мы можем написать:

```
template<class T, class A>
vector<T, A>::vector(size_type n, const T& val, const A& a)
        : alloc(a)
{
  v = alloc.allocate(n);

  try
  {
    uninitialized_fill(v, v+n, val);  // копируем элементы
    space = last = v+n;
  }
  catch(...)
  {
    alloc.deallocate(v, n);           // освобождаем память
    throw;                            // повторно генерируем исключение
  }
}
```

Лично я не стал бы вызывать этот код. В следующем разделе будет показано, как его можно существенно упростить.

Обратите внимание на то, что конструктор повторно генерирует перехваченное исключение. Это делается для того, чтобы тип ***vector*** был прозрачен для исключений и чтобы пользователь мог определить истинную причину исключения. Все контейнеры стандартной библиотеки обладают этим свойством. Прозрачность по отношению к исключениям — часто наилучшая политика для шаблонов и других «тонких» слоев программного обеспечения. Напротив, главные части системы (модули) должны в общем случае брать на себя ответственность за обработку исключений. Разработчик модуля должен перечислить все исключения, которые модуль может сгенерировать. Для этого могут потребоваться группировка исключений (§14.2), отображение исключений низкоуровневых процедур в исключения уровня модуля (§14.6.3) и спецификация исключений (§14.6).

Е.3.2. Явное управление памятью

Опыт показывает, что написание корректного кода, безопасного в контексте исключений, при помощи явного применения *try*-блоков является более трудной задачей, чем это кажется на первый взгляд. Эти трудности даже не всегда оправданы, ибо имеется альтернатива: методология «выделение ресурса есть инициализация» (§14.4) позволяет уменьшить объем кода и сделать его более элегантным. Ключевой ресурс, который требует *vector* — это память, необходимая для хранения его элементов. Если выделить отдельный класс для представления понятия памяти, используемой типом *vector*, можно значительно упростить результирующий код и уменьшить риск допущения утечек памяти:

```
template<class T, class A = allocator<T> >
struct vector_base
{
  A alloc;        // аллокатор
  T* v;           // начало выделенной памяти
  T* space;       // конец последовательности эл-ов (начало свобод. прост-ва)
  T* last;        // конец выделенной памяти

  vector_base(const A& a, typename A::size_type n)
        : alloc(a), v(alloc.allocate(n)), space(v+n), last(v+n) { }

  ~vector_base() {alloc.deallocate(v,last-v);}
};
```

Пока *v* и *last* корректны, *vector_base* может быть уничтожен. Класс *vector_base* имеет дело с памятью для типа *T*, а не с объектами типа *T*. Следовательно, пользователь *vector_base* должен уничтожить все объекты, которые созданы в выделенной *vector_base* памяти, прежде чем уничтожить сам *vector_base*.

Естественно, *vector_base* написан так, что при генерации исключения (копирующим конструктором аллокатора или функцией *allocate*()) никакого объекта *vector_base* не создается и память не теряется.

Нам нужно также уметь обменивать (to swap) друг с другом объекты типа *vector_base*. Умолчательный *swap*(), однако, нашим нуждам не удовлетворяет, поскольку он копирует и уничтожает временные объекты, а так как *vector_base* — это вспомогательный класс специального назначения, которому не придана семантика защищенного копирования, то эти уничтожения приведут к нежелательным побочным эффектам. Как следствие, мы предоставляем специализацию:

```
template<class T> void swap(vector_base<T>& a, vector_base<T>& b)
{
  swap(a.alloc,b.alloc); swap(a.v,b.v);
  swap(a.space,b.space); swap(a.last,b.last);
}
```

Отталкиваясь от *vector_base*, можно определить *vector* следующим образом:

```
template<class T, class A = allocator<T> >
class vector: private vector_base<T,A>
{
  void destroy_elements() {for(T* p = v; p!=space; ++p) p->~T(); space=v;}
```

```
public:
  explicit vector(size_type n, const T& val = T(), const A& = A());

  vector(const vector& a);               // копирующий конструктор
  vector& operator=(const vector& a);    // операция присваивания

  ~vector() {destroy_elements();}

  size_type size() const {return space-v;}
  size_type capacity() const {return last-v;}

  void push_back(const T&);
  // ...
};
```

Деструктор типа ***vector*** явно вызывает деструктор типа ***T*** для каждого элемента. Это означает, что если деструктор элемента генерирует исключение, то уничтожение объекта ***vector*** терпит неудачу. Это просто катастрофа, если все это происходит при раскрутке стека, вызванной исключением, и вызывается функция ***terminate***() (§14.7). В случае обычного уничтожения объекта, генерация исключения в деструкторе приводит к утечке ресурсов и непредсказуемому поведению кода, полагающегося на разумное поведение объектов. Не существует действенной защиты от генерации исключений в деструкторах, так что библиотека не дает никаких гарантий при генерации исключений в деструкторах элементов (§Е.4).

Теперь конструктор можно определить весьма просто:

```
template<class T, class A>
vector<T,A>::vector(size_type n, const T& val, const A& a)
        : vector_base<T,A>(a, n)     // выделяем память для n элементов
{
  uninitialized_fill(v, v+n, val);   // копируем элементы
}
```

Копирующий конструктор отличается лишь использованием ***uninitialized_copy***() вместо ***uninitialized_fill***():

```
template<class T, class A>
vector<T,A>::vector(const vector<T,A>& a)
        : vector_base<T,A>(a.alloc, a.size())
{
  uninitialized_copy(a.begin(), a.end(), v);
}
```

Заметим, что этот стиль конструктора полагается на фундаментальное правило языка: если исключение генерируется в конструкторе, то уже сконструированные подобъекты (такие как поля данных базового класса) будут должным образом уничтожены (§14.4.1). Алгоритм ***uninitialized_fill***() и его собратья (§Е.4.4) предоставляют аналогичную гарантию для частично созданных последовательностей.

Е.3.3. Присваивание

Как обычно, присваивание отличается от конструирования (создания) тем, что нужно позаботиться о старом значении. Рассмотрим прямолинейную реализацию:

```
template<class T, class A>
vector<T,A>& vector<T,A>::operator=(const vector& a)  // дает сильную гарантию (§E.2)
{
  vector_base<T,A> b(a.alloc,a.size());               // получаем память
  uninitialized_copy(a.begin(),a.end(),b.v);          // копируем элементы
  destroy_elements();
  alloc.deallocate(v,last-v);         // освобождаем старую память
  vector_base::operator=(b);          // размещаем новое представление
  b.v = 0;                            // предотвращаем освобождение
  return *this;
}
```

Это присваивание безопасно, но оно повторяет много кода из конструкторов и деструктора. Избежать этого можно следующим образом:

```
template<class T, class A>
vector<T,A>& vector<T,A>::operator=(const vector& a)  // дает сильную гарантию (§E.2)
{
  vector temp(a);                          // копируем a
  swap<vector_base<T,A> >(*this,temp);     // обмен представлениями
  return *this;
}
```

Старые элементы уничтожаются деструктором переменной *temp*, а память, использованная для их хранения, освобождается деструктором класса *vector_base* при вызове деструктора для *temp*.

Производительность этих двух версий должна быть примерно одинаковой. По сути дела, это лишь два разных способа определения одних и тех же операций, но вторая версия короче и не реплицирует код из родственных функций типа *vector*, так что ее применение менее подвержено ошибкам и сопровождать ее легче.

Обратите внимание на отсутствие традиционной проверки на самоприсваивание (§10.4.4). Показанные реализации присваивания сначала создают копию, а затем меняют местами представления. Очевидно, что при этом корректно обрабатывается и случай самоприсваивания. Я решил, что эффективность, достигаемая наличием проверки редкого случая самоприсваивания, не стоит потерь в общем случае, когда присваивается отличающийся *vector*.

В обоих вариантах упущены две возможности значительной оптимизации кода:

1. Если емкость вектора, которому присваивается значение, достаточно велика для того, чтобы вместить все элементы присваиваемого вектора, то нет необходимости в выделении новой памяти.
2. Присваивание элементов может оказаться эффективнее их уничтожения с последующим конструированием.

Реализуя указанные оптимизации, получаем:

```
template<class T, class A>
vector<T,A>& vector<T,A>::operator=(const vector& a)  // дает базовую гарантию (§E.2)
{
  if(capacity() < a.size())      // выделяем память под новое представление:
  {
    vector temp(a);              // копируем a
    swap<vector_base<T,A> >(*this,temp);          // обмениваем представления
```

```
    return *this;
  }

  if(this == &a) return *this;                  // защита от самоприсваивания (§10.4.4)

  size_type sz = size();
  size_type asz = a.size();

  if(asz<=sz)
  {
    copy(a.begin(),a.begin()+asz,v);            // присваиваем старые элементы
    for(T* p = v+asz; p!=space; ++p) p->~T();   // уничтожаем лишние эл-ты (§10.4.11)
  }
  else
  {
    copy(a.begin(),a.begin()+sz, v);            // присваиваем старые элементы
    uninitialized_copy(a.begin()+sz,a.end(),space); // конструируем дополнит. эл-ты
  }

  space = v+asz;
  return *this;
}
```

Выполненная оптимизация не бесплатна. Алгоритм ***copy***() (§18.6.1) не предоставляет сильной гарантии безопасности при исключениях. Он не гарантирует, что оставит свой целевой объект нетронутым, если в процессе копирования будет сгенерировано исключение. Таким образом, если ***T***::***operator=***() сгенерирует исключение в процессе работы ***copy***(), то вектор, которому присваивается значение другого вектора, не обязательно будет равен последнему, или не обязательно останется в исходном состоянии. Например, первые пять элементов окажутся копиями элементов присваиваемого вектора, а остальные останутся неизменными. Также вполне вероятно, что тот элемент, который копировался тогда, когда ***T***::***operator=***() сгенерировал исключение, останется со значением, которое и не равно старому значению, и не равно значению копировавшегося элемента. И все же, если ***T***::***operator=***() оставляет свои операнды при генерации исключения в действительном состоянии, то ***vector*** также будет находиться в действительном состоянии, пусть и не в том, которое мы ожидали.

Стандарт не требует, чтобы каждый аллокатор поддерживал присваивание (§19.4.3); он также не указывает в точности, когда аллокаторы копируются. Здесь я копирую аллокатор всегда, когда копируются элементы, память под которые была выделена этим аллокатором.

Присваивание для vector из стандартной библиотеки обеспечивает такую же слабую гарантию безопасности исключений вместе с выигрышем в производительности. Присваивание для стандартного типа ***vector*** обеспечивает лишь базовую гарантию безопасности исключений, что отвечает требованиям со стороны большинства пользователей. Еще раз подчеркнем, что ***vector*** не предоставляет при этом сильной гарантии (§E.2). Если же вам нужно, чтобы в случае присваивания для типа vector левый операнд оставался в исходном (то есть неизменном) состоянии при возникновении исключения, то вам придется либо воспользоваться такой реализацией стандартной библиотеки, которая предоставляет в этом случае сильную гарантию, либо написать свою собственную версию операции присваивания. Например:

```
template<class T, class A>
void safe_assign(vector<T,A>& a, const vector<T,A>& b)
{
  vector<T,A> temp(b.get_allocator());
  temp.reserve(b.size());

  for(typename vector<T,A>::iterator p = b.begin(); p!=b.end(); ++p)
    temp.push_back(*p);
  swap(a,temp);
}
```

При нехватке памяти для создания ***temp*** размером в ***b.size***() элементов сгенерируется исключение ***std::bad_alloc*** до того, как будут внесены изменения в ***a***. Точно так же, если ***push_back***() терпит неудачу по любой причине, вектор a останется нетронутым, потому что мы применяем ***push_back***() к ***temp***, а не к ***a***. В этом случае элементы ***temp***, созданные функцией ***push_back***(), будут уничтожены до того как исключение, вызвавшее сбой в работе, будет сгенерировано повторно.

Функция ***swap***() не копирует элементы векторов. Она просто меняет местами их поля данных; то есть она обменивает их подобъекты ***vector_base*** (§Е.3.2). Следовательно, она не генерирует исключений, даже если операции на элементах могли бы это делать (§Е.4.3). В итоге, ***safe_assign***() не делает паразитных копий элементов и демонстрирует приличную эффективность.

Как это нередко бывает, имеются альтернативы для очевидных реализаций. Мы можем позволить библиотеке выполнять для нас копирование во временную переменную:

```
template<class T, class A>
void safe_assign(vector<T,A>& a, const vector<T,A>& b)
{
  vector<T,A> temp(b);      // копируем элементы b в temp
  swap(a,temp);
}
```

На самом деле, можно воспользоваться и передачей параметров по значению (§7.2.):

```
template<class T, class A>
void safe_assign(vector<T,A>& a, vector<T,A> b)   // b передается по значению
{
  swap(a,b);
}
```

Е.3.4. Метод push_back()

В контексте безопасности исключений, ***push_back***() аналогична присваиванию в том смысле, что мы должны позаботиться о неизменности ***vector*** в случае неудачи с добавлением нового элемента:

```
template<class T, class A>
void vector<T,A>::push_back(const T& x)
{
  if(space == last)                          // закончилось свободное пространство:
  {
```

```
        vector_base b(alloc,size() ? 2*size() : 2);   // удваиваем размер
        uninitialized_copy(v,space,b.v);
        new (b.space) T (x);                         // помещаем копию x в *b.space (§10.4.11)
        ++b.space;
        destroy_elements();
        swap<vector_base<T,A> >(b,*this);            // обмениваем представления
        return;
    }
    new (space) T(x);                                // помещаем копию x в *space (§10.4.11)
    ++space;
}
```

Разумеется, копирующий конструктор, использованный для инициализации ***space**, может сгенерировать исключение. Если это произойдет, значение ***vector*** останется неизменным, а ***space*** не увеличится. В этом случае элементы ***vector*** остались на своих местах в памяти, так что ссылающиеся на них итераторы действительны. Таким образом, эта реализация предоставляет сильную гарантию того, что исключение, сгенерированное аллокатором или даже копирующим конструктором, предоставленным пользователем, оставит vector неизменным. Стандартная библиотека предоставляет именно такую гарантию для ***push_back***() (§E.4.1).

Обратите внимание на отсутствие ***try***-блока (кроме скрытого в ***uninitialized_copy***()). За счет тщательного упорядочения операций удалось гарантировать неизменность vector при возникновении исключений.

Подход, при котором безопасность исключений достигается путем упорядочения кода и следования методике «выделение ресурса есть инициализация» (§14.4), часто оказывается более элегантным и эффективным, чем явная обработка ошибок с использованием ***try***-блоков. Однако при неправильном упорядочении кода возникает больше проблем с достижением безопасности исключений, чем из-за отсутствия особого кода обработки ошибок. Основное правило упорядочения кода — не уничтожать старую информацию до того, как будут полностью созданы новые данные, присваивание которых не подвержено риску возникновения исключений.

Исключения создают почву для неожиданностей по части изменений потока управления (то есть последовательности выполнения операторов программы). В простых локальных фрагментах кода, в которых доминирует линейная последовательность вызовов ***operator=***(), ***safe_assign***() и ***push_back***(), возможностей для подобного рода сюрпризов мало. Однако в больших функциях, содержащих множество циклов и условных операторов, прогноз сделать труднее. Добавление ***try***-блоков лишь увеличивает сложность управления, и может служить источником недоразумений и ошибок (§14.4). Поэтому подход с упорядочением кода и следованием методике «выделение ресурса есть инициализация» имеет и здесь преимущество: упрощается контроль за потоком исполнения кода программы. Простой, элегантный код всегда легче понять и отладить.

Отметим, что рассмотренная нами реализация vector приведена лишь с целью иллюстрации трудностей, которые могут породить исключения, а также методов их преодоления. Стандарт не требует от реализаций библиотеки следовать этим рекомендациям. Настоящие гарантии стандарта рассмотрены ниже в §E.4.

Е.3.5. Конструкторы и инварианты

С точки зрения безопасности исключений другие операции для типа ***vector*** или эквивалентны уже рассмотренным (поскольку приобретают и освобождают ресурсы схожим образом), или тривиальны (не требуют соблюдения действительных состояний участвующих в них объектов). Эти операции для большинства классов составляют большую часть кода, а легкость их написания критически зависит от среды, устанавливаемой конструктором для их исполнения. Говоря иначе, все зависит от качества выбора инварианта класса (§24.3.7.1). Изучение тривиальных функций класса поможет нам понять, что делает инвариант класса хорошим, и как нужно писать конструкторы, чтобы они устанавливали хорошие инварианты.

Такие операции, как индексирование векторов (§16.3.3), писать просто, поскольку они могут положиться на инвариант, устанавливаемый конструкторами и поддерживаемый всеми функциями, которые захватывают или освобождают ресурсы. В частности, операция индексирования может положиться на то, что поле ***v*** ссылается на массив элементов:

```
template<class T, class A>
T& vector<T,A>::operator[] (size_type i)
{
  return v[i];
}
```

Крайне важно, чтобы конструкторы, захватывая ресурсы, устанавливали простой инвариант. Чтобы понять, почему это так важно, рассмотрим альтернативное определение ***vector_base***:

```
template<class T, class A = allocator<T> >
class vector_base
{
public:
  A alloc;
  T* v;
  T* space;
  T* last;

  vector_base(const A& a, typename A::size_type n)
          : alloc(a), v(0), space(0), last(0)
  {
    v = alloc.allocate(n);
    space = last = v+n;
  }

  ~vector_base() {if(v) alloc.deallocate(v,last-v);}
};
```

Здесь я создаю ***vector_base*** в две стадии: сначала я создаю безопасное (надежное) состояние с ***v***, ***space*** и ***last*** равными нулю, и только после этого пробую выделять память. Все это сделано из необоснованного опасения, что во время выделения памяти под элементы произойдет исключение, и конструктор оставит после себя частично созданный объект. Однако это опасение необосновано, так как невозможно оставить после себя частично созданный объект и позднее обратиться к нему — правила для статических и автоматических объектов, объектов-членов класса и элементов контейнеров

стандартной библиотеки предотвращают такую возможность. Это было возможно для «достандартных» вариантов библиотеки, которые использовали операцию «размещающее new» (§10.4.11) для создания объектов в контейнерах, разработанных без учета безопасности исключений. Привычки, однако, изживаются с трудом.

Отметим, что предпринятая нами попытка написать код «побезопаснее», лишь усложняет инвариант для класса: уже не гарантируется, что *v* указывает на выделенную память. Теперь поле *v* может быть равно *0*, а это немедленно вызывает проблемы, так как в стандартной библиотеке для аллокаторов не гарантируется безопасное освобождение памяти по нулевому указателю (§19.4.1). В этом отношении аллокаторы отличаются от операции ***delete*** (§6.2.6). В итоге, пришлось предусмотреть в деструкторе соответствующую проверку. Кроме того, каждый элемент сначала инициализируется нулем, после чего ему присваивается значение. Стоимость этой лишней работы может оказаться существенной для типов элементов, у которых операция присваивания нетривиальна (вроде ***string*** или ***list***).

Рассмотренный двухступенчатый конструктор не так уж необычен. Иногда этот стиль делают явным, оставляя конструктору лишь простую и безопасную инициализацию, обеспечивающую корректное состояние объекта для его потенциального уничтожения. Реальное же создание объекта оставляется функции ***init***(), которую пользователь должен вызвать явно. Например:

```
template<class T>       // архаичный стиль
class vector_base
{
public:
  T* v;
  T* space;
  T* last;

  vector_base() : v(0), space(0), last(0) {}
  ~vector_base() {free(v);}

  bool init(size_t n)      // возвращает true в случае успешной инициализации
  {
    if(v = (T*)malloc(sizeof(T)*n))
    {
      uninitialized_fill(v,v+n,T());
      space = last = v+n;
      return true;
    }
    return false;
  }
};
```

Предполагается, что ценность данного стиля состоит в том, что:

1. Конструктор не генерирует исключений, а успешность отработки функции ***init***() можно проконтролировать обычными (то есть без исключений) средствами.
2. Существует тривиальное действительное состояние. При возникновении проблем операция может придать это состояние объекту.
3. Выделение ресурсов откладывается до момента, когда на самом деле возникает нужда в полностью инициализированном объекте.

Все эти пункты изучаются в последующих разделах, где показывается, почему эта двухступенчатая методика создания объектов не приносит ожидаемых выгод. Более того, она может стать источником проблем.

Е.3.5.1. Применение функции init()

Первый пункт (использование функции *init*() вместо полноценного конструктора) — это ложная идея. Полноценные конструкторы и обработка исключений — намного более общий и систематический способ работы с ресурсами и ошибками инициализации (§14.1, §14.4). Применение функции *init*() — это пережиток эпохи «достандартного» С++ в отсутствие механизма исключений.

Тщательно составленный код обоих стилей более-менее эквивалентен друг другу. Рассмотрим следующие примеры:

```
int f1 (int n)
{
  vector<X> v;
  // ...
  if(v.init(n))
  {
    // используем v вектор из n элементов
  }
  else
  {
    // улаживаем проблему
  }
}
```

и

```
int f2 (int n)
try
{
  vector<X> v(n);
  // ...
  // используем v вектор из n элементов
}
catch (...)
{
  // улаживаем проблему
}
```

При всем при этом, использование отдельной функции *init*() — это возможность:

1. Забыть вызвать *init*() (§10.2.3).
2. Забыть проверить успешность отработки *init*().
3. Вызвать *init*() более одного раза.
4. Забыть, что *init*() может сгенерировать исключение.
5. Использовать объект до вызова *init*().

В хорошей реализации С++ функция *f2*() чуть быстрее функции *f1*(), поскольку она избегает в общем случае проверки.

Е.3.5.2. Полагаемся на действительное состояние по умолчанию

Второй пункт (простое действительное состояние по умолчанию) в общем случае правилен, но для vector он ведет к ненужным расходам: теперь ***vector_base*** может иметь ***v==0***, так что в реализации типа vector нужно всюду защищаться от этого. Например:

```
template<class T>
T& vector<T>::operator[] (size_t i)
{
  if(v) return v[i];
  // улаживаем проблему
}
```

Наличие возможности ***v==0*** делает стоимость операции индексирования без проверки диапазона равной стоимости операции индексирования с проверкой диапазона индексов:

```
template<class T>
T& vector<T>::at(size_t i)
{
  if(i<v.size()) return v[i];
  throw out_of_range("vector index");
}
```

Главное здесь то, что допуская возможность ***v==0***, я усложнил базовый инвариант для ***vector_base***, а как следствие усложнился и таковой для ***vector***. В конечном итоге, весь код ***vector_base*** и ***vector*** усложнился в связи с этим, что является источником возможных ошибок, усложняет поддержку кода и увеличивает накладные расходы на этапе выполнения. Отметим, что условные операторы могут оказаться неожиданно дорогими на современных компьютерных архитектурах. Если производительность критически важна, то ключевые операции (вроде индексирования вектора) следует реализовывать без условных операторов.

Интересно, что первоначальное определение ***vector_base*** уже содержало легко конструируемое действительное состояние: объект ***vector_base*** не мог существовать без успешного выделения памяти. Из-за этого реализация ***vector*** могла содержать следующую «функцию аварийного выхода»:

```
template<class T, class A>
void vector<T,A>::emergency_exit()
{
  space = v;      // устанавливаем размер *this в 0
  throw Total_failure();
}
```

Это немного грубовато, так как при этом не удается вызвать деструкторы элементов и освободить в них память, отведенную под поля данных базового типа ***vector_base***. То есть не обеспечивается базовая гарантия (§Е.2). Если же мы намерены доверять значениям ***v*** и ***space*** и деструкторам элементов, то мы можем избежать потенциальных утечек ресурсов:

```
template<class T, class A>
void vector<T,A>::emergency_exit()
```

```
{
  destroy_elements () ;      // очистка
  throw Total_failure () ;
}
```

Отметим, что стандартный ***vector*** спроектирован настолько аккуратно, что он минимизирует проблемы, вызываемые двухфазным конструированием. Функция ***init***() приблизительно эквивалентна функции ***resize***(), и почти везде возможность ***v==0*** закрыта проверками ***size***()==***0***. Отрицательные эффекты двухфазного конструирования становятся особо заметными в случае классов, захватывающих важные ресурсы, такие как сетевые соединения и файлы. Эти классы редко являются частью более общей среды, в рамках которой формулировались бы способы их реализации и использования, как это имеет место для контейнера ***vector*** в стандартной библиотеке языка C++. Обычно проблемы усиливаются еще и из-за того, что понятия прикладных областей плохо соответствуют описанию реально захватываемых компьютерных ресурсов, необходимых для их реализации. Немного существует классов, концепции которых отображаются на системные ресурсы столь непосредственно, как это имеет место для контейнера ***vector***.

Идея «безопасного состояния» в принципе хорошая. Если мы не можем поместить объект в безопасное состояние без риска генерации исключения в процессе выполнения операции, у нас и в самом деле проблемы. Однако «безопасное состояние» должно вытекать из семантики класса, а не быть артефактом реализации, усложняющим инвариант этого класса.

Е.3.5.3. Отложенное выделение ресурсов

Подобно второму пункту (§Е.2), третий пункт (отложить выделение ресурса до момента, когда он на самом деле потребуется) неправильно применяет хорошую идею, так что издержки есть, а выгод нет. Во многих случаях, особенно для контейнеров вроде ***vector***, лучший способ отложить выделение ресурсов — это отложить создание объекта. Рассмотрим неоптимальное использование контейнера ***vector***:

```
void f(int n)
{
  vector<X> v(n) ;      // создаем n умолчательных значений типа X
  // ...
  v[3] = X(99) ;        // реальная "инициализация" v[3]
  // ...
}
```

Создание *X* с единственной целью присвоить ему позже новое значение, является расточительной идеей, особенно если присваивание для типа *X* не дешево. Поэтому двухфазное конструирование *X* может показаться привлекательным. Например, если тип *X* сам является вектором, то можно было бы рассмотреть двухфазное создание vector с целью эффективного создания пустых векторов. Однако создание умолчательных (пустых) векторов и так эффективно, поэтому усложнение реализации с учетом частного случая пустого вектора представляется бесполезным. Вообще, освобождение кода конструктора элементов от сложной инициализации редко когда является наилучшим решением для реализации фиктивной инициализации. Вместо этих ухищрений пользователь может создавать элементы строго по необходимости. Например:

```
void f2 (int n)
{
  vector<X> v;              // создаем пустой vector
  // ...
  v.push_back (X(99)) ;     // конструируем элементы по необходимости
  // ...
}
```

Подводим итог: двухфазное конструирование объектов приводит к более сложным инвариантам, и, как правило, к менее элегантному, чреватому ошибками коду, более сложному в сопровождении. Где это возможно, следует отдавать предпочтение коду, базирующемуся на конструкторах, а не на ***init***()-функциях, поскольку первый подход поддерживается языковыми средствами. То есть ресурсы должны выделяться в конструкторах, если только отложенное выделение ресурсов не диктуется унаследованной семантикой класса.

Е.4. Гарантии стандартных контейнеров

Если операция стандартной библиотеки генерирует исключение, она может быть уверена в том, что ее объекты-операнды остаются в действительном состоянии. Например, ***at***(), генерирующая исключение ***out_of_range*** для ***vector*** (§16.3.3), не создает проблем с обеспечением безопасности исключений для контейнера ***vector***. Разработчик операции ***at***() не сомневается, что в момент генерации исключения ***vector*** находится в хорошо определенном состоянии. Трудности для разработчиков библиотеки, ее пользователей, а также людей, пытающихся разобраться в коде, начинаются тогда, когда исключение генерирует функция, предоставленная пользователем.

Контейнеры стандартной библиотеки предоставляют базовую гарантию (§Е.2): соблюдаются основные инварианты библиотеки и нет утечек ресурсов до тех пор, пока пользовательский код ведет себя как надо, то есть пользовательские «операции» не оставляют элементы контейнеров в недействительных состояниях и не генерируют исключений из деструкторов. Под «операциями» я здесь понимаю операции, используемые реализацией стандартной библиотеки, такие как конструкторы, операции присваивания, деструкторы и операции над итераторами (§Е.4.4).

Программисту относительно легко добиться того, чтобы эти операции соответствовали ожиданиям библиотеки. Даже бесхитростно написанный код часто фактически соответствует требованиям библиотеки. Следующие типы определенно удовлетворяют требованиям стандартной библиотеки к типам контейнерных элементов:

1. Встроенные типы — включая указатели.
2. Типы без пользовательских операций.
3. Классы с операциями, не генерирующими исключений и не оставляющими операнды в недействительных состояниях.
4. Классы с деструкторами, не генерирующими исключений, и с операциями (которые использует стандартная библиотека — конструкторы, присваивания, <, ==, ***swap***() и т.д.), для которых можно легко убедиться, что они не оставляют операнды в недействительных состояниях.

В любом случае нужно также убедиться, что нет утечек ресурсов. Например:

```
void f(Circle* pc, Triangle* pt, vector<Shape*>& v2)
{
  vector<Shape*> v(10);          // создает vector или генерирует bad_alloc
  v[3] = pc;                     // нет генерации исключений
  v.insert(v.begin()+4,pt);      // или внедряет pt, или не изменяет v
  v2.erase(v2.begin()+3);        // или удаляет v2[3], или не изменяет v2
  v2 = v;                        // копирует v или не изменяет v2
  // ...
}
```

Когда функция *f*() закончит работу, ***v*** будет надлежащим образом уничтожена, а ***v2*** останется в действительном состоянии. Из данного фрагмента не ясно, кто отвечает за удаление ***pc*** и ***pt***. Если это ответственность функции *f*(), то она может либо перехватывать исключения и выполнять необходимые удаления, либо присваивать указатели локальным переменным типа ***auto_ptr***.

Более интересный вопрос заключается в следующем: когда библиотечные операции предлагают сильную гарантию (то есть либо успешно завершаются, либо оставляют операнды нетронутыми)? Например:

```
void f(vector<X>& vx)
{
  vx.insert(vx.begin()+4,X(7));  // добавляем элемент
}
```

Вообще говоря, операции *X* и аллокатор у ***vector<X>*** могут генерировать исключения. Что можно сказать об элементах ***vx***, когда *f*() завершается по исключению? Базовая гарантия утверждает, что нет утечек памяти, и что элементы ***vx*** находятся в действительных состояниях. Но остается вопрос — изменился ли при этом ***vx***? Не добавилось ли умолчательных значений *X*? Мог ли элемент быть удален из-за того, что для ***insert***() это единственный способ восстановиться с соблюдением базовой гарантии? Иногда не достаточно знать, что контейнер находится в хорошем состоянии; часто нужно знать, что это за состояние. После перехвата исключения нам обычно интересно знать, являются ли значения элементов теми, что мы ожидаем, или же нужно приступать к восстановлению после ошибок.

Е.4.1. Вставка и удаление элементов

Вставка элемента в контейнер и удаление элемента из контейнера — это очевидные примеры операций, которые в случае возникновения исключений могут оставить контейнер в непредсказуемом состоянии. Причина заключается в том, что вставка или удаление активизируют множество операций, которые могут генерировать исключения:

1. Новое значение копируется в контейнер.
2. Нужно уничтожить элемент, удаленный из контейнера.
3. Под новый элемент требуется выделить память.
4. Элементы vector или deque нужно переместить в новое место в памяти.
5. Ассоциативные контейнеры вызывают операции сравнения для элементов.
6. Вставки и удаления включают операции с итераторами.

В каждом из перечисленных случаев могут генерироваться исключения.

Если деструктор генерирует исключение, не дается вообще никаких гарантий (§Е.2), так как в этом случае они обошлись бы непозволительно дорого. Но от генерации исключений другими операциями, предоставляемыми пользователем, библиотека защищает и себя, и своих пользователей.

При манипуляциях со связными структурами данных, вроде ***list*** или ***map***, элементы могут добавляться или удаляться, не оказывая влияния на другие элементы этих контейнеров. Иначе обстоит дело для контейнеров вроде ***vector*** или ***deque***, элементы которых распределены в памяти непрерывно; здесь иногда приходится перемещать элементы на новые места.

Дополнительно к базовой гарантии стандартная библиотека предоставляет сильную гарантию для нескольких операций, вставляющих и удаляющих элементы. Поскольку контейнеры, реализованные в виде связных структур данных, ведут себя иначе, чем контейнеры с непрерывным размещением элементов в памяти, стандарт обеспечивает слегка различающиеся гарантии для разных видов контейнеров:

1. Гарантии для контейнеров ***vector*** (§16.3) и ***deque*** (§17.2.3):
   - Если исключение генерируется в функциях ***push_back***() или ***push_front***(), то они оставляют контейнер нетронутым.
   - Если исключение генерируется в функции ***insert***(), но не копирующим конструктором или операцией присваивания типа элемента, то контейнер остается нетронутым.
   - Функция ***erase***() не генерирует исключений, кроме случаев возникновения исключений в копирующем конструкторе или в операции присваивания типа элемента.
   - Функции ***pop_back***() и ***pop_front***() не генерируют исключений.
2. Гарантии для контейнера list (§17.2.2):
   - Если исключение генерируется в функциях ***push_back***() или ***push_front***(), то они оставляют контейнер нетронутым.
   - Если исключение генерируется в функции ***insert***(), то контейнер остается нетронутым.
   - Функции ***erase***(), ***pop_back***(), ***pop_front***(), ***splice***() и ***reverse***() не генерируют исключений.
   - Функции-члены контейнера ***list***, такие как ***remove***(), ***remove_if***(), ***unique***(), ***sort***() и ***merge***(), не генерируют исключений, кроме случаев возникновения исключений в предикате или функции сравнения.
3. Гарантии для ассоциативных контейнеров (§17.4):
   - Если исключение генерируется операцией ***insert***() при вставке единственного элемента, то контейнер остается нетронутым.
   - Функция ***erase***() не генерирует исключений

Обратите внимание, что если для операций на контейнере дается строгая гарантия, то при генерации исключения все итераторы, указатели и ссылки на элементы остаются действительными.

Эти правила могут быть объединены в таблицу:

Гарантии операций контейнеров				
	vector	*deque*	*list*	*map*
***clear*()**	не генерирует (копирование)	не генерирует (копирование)	не генерирует	не генерирует
***erase*()**	не генерирует (копирование)	не генерирует (копирование)	не генерирует	не генерирует
1-элементный ***insert*()**	сильная (копирование)	сильная (копирование)	сильная	сильная
N-элементный ***insert*()**	сильная (копирование)	сильная (копирование)	сильная	базовая
***merge*()**	—	—	не генерирует (сравнение)	—
***push_back*()**	сильная	сильная	сильная	—
***push_front*()**	—	сильная	сильная	—
***pop_back*()**	не генерирует	не генерирует	не генерирует	—
***pop_front*()**	—	не генерирует	не генерирует	—
***remove*()**	—	—	не генерирует (сравнение)	—
***remove_if*()**	—	—	не генерирует (предикат)	—
***reverse*()**	—	—	не генерирует	—
***splice*()**	—	—	не генерирует	—
***swap*()**	не генерирует	не генерирует	не генерирует	не генерирует (копир-е сравнения)
***unique*()**	—	—	не генерирует (сравнение)	—

Где гарантия требует отсутствия исключений в некоторой пользовательской операции, там эта операция указана в круглых скобках под строкой гарантии. Эти требования точно сформулированы в тексте, предшествующем таблице.

Функции ***swap*()** отличаются от других упомянутых функций тем, что не являются функциями-членами. Гарантия для ***clear*()** вытекает из таковой для ***erase*()** (§16.3.6). В таблице перечислены гарантии, предоставляемые в дополнение к базовой гарантии. Поэтому в таблицу не вошли такие операции, как ***reverse*()** и ***unique*()** для ***vector***, поскольку они предоставляются как алгоритмы для любых последовательностей без дополнительных гарантий.

«Почти контейнер» ***basic_string*** (§17.5, §20.3) предоставляет базовую гарантию для всех операций. Стандарт также гарантирует, что ***erase*()** и ***swap*()** для этого типа не генерируют исключений, и что на функции ***insert*()** и ***push_back*()** распространяется сильная гарантия.

Помимо неизменности контейнера, операция, предоставляющая сильную гарантию, также оставляет действительными все итераторы, указатели и ссылки. Например:

```
void update (map<string,X>& m, map<string,X>::iterator current)
{
 X x;
 string S;

 while (cin>>s>>x)
  try
  {
   current = m.insert(current,make_pair(s,x));
  }
  catch (...)
  {
   // здесь current по-прежнему соответствует текущему элементу
  }
}
```

Е.4.2. Гарантии и компромиссы

Затейливая смесь дополнительных гарантий отражает сущность реализации. Программисты предпочитают сильную гарантию с возможно меньшим числом оговорок, но они же настаивают, чтобы каждая операция стандартной библиотеки была как можно более эффективна. Оба пожелания разумны, но для многих операций невозможно удовлетворить их одновременно. Чтобы дать лучшее представление о компромиссах, я исследую способы добавления единственного или нескольких элементов к контейнерам ***list***, ***vector*** и ***map***.

Рассмотрим сначала добавление единственного элемента к ***list*** или ***vector***. Как всегда, ***push_back***() обеспечивает простейший способ для этого:

```
void f(list<X>& lst, vector<X>& vec, const X& x)
{
 try
 {
  lst.push_back(x);        // добавляем элемент к list
 }
 catch (...)
 {
  // lst не изменился
  return;
 }

 try
 {
  vec.push_back(x);        // добавляем элемент к vector
 }
 catch (...)
 {
  // vec не изменился
```

```
    return;
  }
  // lst и vec получили по новому элементу со значением x
}
```

Обеспечение сильной гарантии в таких случаях дается легко и дешево. Она весьма полезна, поскольку представляет собой полностью безопасный в контексте исключений способ добавления элементов. Однако ***push_back()*** не определяется для ассоциативных контейнеров — у контейнера map отсутствует ***back()***. В конце концов, последний элемент ассоциативного контейнера определяется отношением порядка, а не позицией.

Гарантии для ***insert()*** несколько более сложные. Причина состоит в том, что иногда функции ***insert()*** нужно помещать элемент куда-то в «середину контейнера». Это не проблема для связанных структур данных, таких как ***list*** или ***map***. В то же время, при наличии свободного зарезервированного пространства у контейнера типа ***vector***, очевидная реализация ***vector<X>::insert()*** копирует элементы, расположенные после точки вставки, чтобы освободить место. Это эффективно, но нет простого способа восстановления ***vector***, если копирующий конструктор типа ***X*** или его операция присваивания генерируют исключение (см. §Е.8[10-11]). Следовательно, ***vector*** обеспечивает гарантию лишь при условии, что копирующие операции элементов не генерируют исключений. А контейнеры ***list*** и ***map*** не нуждаются в этом условии; они могут запросто привязать новые элементы уже после выполнения любого копирования.

Для примера предположим, что копирующие операции типа ***X*** генерируют исключение X::cannot_copy, если не получается успешно создать копию:

```
void f(list<X>& lst, vector<X>& vec, map<string,X>& m, const X& x, const string& s)
{
  try
  {
    lst.insert(lst.begin(),x);      // добавляем элемент к list
  }
  catch(...)
  {
    // lst не изменился
    return;
  }

  try
  {
    vec.insert(vec.begin(),x);      // добавляем элемент к vector
  }
  catch(X::cannot_copy)
  {
    // oops: vec может содержать, а может и не содержать новый элемент
    return;
  }
  catch(...)
  {
    // vec не изменился
```

```
    return;
  }
  try
  {
    m.insert(make_pair(s,x));     // добавляем элемент к map
  }
  catch(...)
  {
    // m не изменился
    return;
  }
  // lst и vec получили по новому элементу со значением x
  // m получил элемент со значением (s,x)
}
```

Если исключение *X::cannot_copy* перехвачено, то в этот момент новый элемент либо уже вставлен в *vec*, либо не вставлен. В первом случае объект находится в действительном состоянии, но каково его значение — неизвестно. Возможно, что после генерации исключения *X::cannot_copy* некоторый элемент окажется мистическим образом продублированным (см. §E.8[11]). Или же *insert*() реализована таким образом, чтобы удалять некоторые хвостовые элементы с целью быть уверенным в том, что в контейнере не оставлены недействительные элементы.

К сожалению, обеспечение сильной гарантии для *insert*() в контейнере vector в отсутствии оговорок про копирующие операции типа элементов непозволительно. Цена полной защиты от исключений при перемещении элементов вектора будет весьма значительной в сравнении со случаем предоставления лишь базовой гарантии.

Типы элементов, генерирующие исключения в процессе выполнения копирующих операций, не столь уж и редки. Вот примеры из самой стандартной библиотеки — *vector<string>*, *vector<vector<double> >* и *map<string,int>*.

Контейнеры *list* и vector предоставляют одинаковые гарантии при вставке одного или нескольких элементов. Причина состоит в том, что реализация *insert*() для *list* и *vector* применяет одну и ту же стратегию вставки как одного, так и нескольких элементов. В то же время map предоставляет сильную гарантию для одноэлементного *insert*(), и лишь базовую гарантию для многоэлементных *insert*(). Легко реализовать вставку одного элемента в map с обеспечением сильной гарантии. Однако типичная реализация многоэлементной вставки в map состоит в том, чтобы вставлять элементы один за другим, а в этом случае трудно обеспечить сильную гарантию. Проблема заключается в том, что нет простого способа отката предыдущих успешных вставок в момент, когда терпит неудачу очередная вставка.

Если мы потребуем сильной гарантии, что либо все элементы успешно вставлены, либо операция ни на что не повлияла, мы можем создать новый контейнер с последующим выполнением *swap*():

```
template<class C, class Iter>
void safe_insert(C& c, typename C::const_iterator i, Iter begin, Iter end)
{
  C tmp(c.begin(),i);                    // копируем начальные элементы
```

```
    copy (begin, end, inserter (tmp, tmp.end () ) ) ;    // копируем новый элемент
    copy (i, c.end () , inserter (tmp, tmp.end () ) ) ;  // копируем "хвостовые" элементы

    swap (c, tmp) ;
}
```

Как всегда, этот код может повести себя некорректно, если деструктор элемента сгенерирует исключение. В то же время, если исключение генерируется копирующими операциями элемента, то аргумент-контейнер остается неизменным.

Е.4.3. Функция swap()

Подобно копирующим конструкторам и присваиваниям операции *swap*() необходимы для многих стандартных алгоритмов и часто предоставляются пользователем. Например, *sort*() и *stable_sort*() обычно переупорядочивают элементы с помощью *swap*(). Таким образом, если функция *swap*() генерирует исключение в момент перестановки элементов контейнера, контейнер может так и остаться с непереставленными элементами или с продублированными элементами, а не с парой элементов, поменявшихся местами.

Рассмотрим очевидное определение функции *swap*() из стандартной библиотеки (§18.6.8):

```
template<class T> void swap (T& a, T& b)
{
  T tmp = a;
  a = b;
  b = tmp;
}
```

Ясно, что *swap*() не генерирует исключений, если только этого не делают копирующие операции типа элемента.

За одним незначительным исключением для ассоциативных контейнеров, гарантируется, что функции *swap*() стандартных контейнеров не генерируют исключений. Как правило, обмен контейнерами выполняется путем обмена структурами данных, играющими роль дескрипторов доступа к элементам (§13.5, §17.1.3). Поскольку сами элементы не перемещаются в памяти, то конструкторы и операции присваивания для элементов не вызываются и, соответственно, исключений не генерируют. Кроме того, стандарт гарантирует, что никакая функция *swap*() стандартной библиотеки не портит значений ссылок, указателей и итераторов, ссылающихся на элементы обмениваемых контейнеров. В итоге, остается единственный потенциальный источник исключений: в ассоциативных контейнерах объект сравнения (§17.1.4.1) копируется как часть дескриптора. Его копирующий конструктор и операция присваивания могут генерировать исключение при вызове функций *swap*() для стандартных контейнеров. По счастью, операции копирования объектов сравнения обычно тривиальны и у них нет поводов генерировать исключения.

Пользовательские функции *swap*() должны предоставлять те же самые гарантии. Этого несложно добиться, если программист обменивает дескрипторы, а не тупо копирует всю информацию, на которую дескрипторы ссылаются (§13.5, §16.3.9, §17.1.3).

Е.4.4. Инициализация и итераторы

Выделение памяти под элементы и инициализация этой памяти — основные части любой реализации контейнеров (§Е.3). Поэтому гарантируется, что стандартные алгоритмы для конструирования объектов в неинициализированной памяти — ***uninitialized_fill***(), ***uninitialized_fill_n***() и ***uninitialized_copy***() (§19.4.4) — не оставляют после себя никаких сконструированных объектов в случае генерации ими исключений. Таким образом, эти алгоритмы предоставляют сильную гарантию (§Е.2). Так как иногда при этом приходится уничтожать элементы, то требование к деструкторам элементов не генерировать исключений является для этих алгоритмов существенным (см. §Е.8[14]). Кроме того, требуется, чтобы итераторы, передаваемые этим алгоритмам в качестве аргументов, вели себя корректно: они сами должны быть действительными, должны ссылаться на действительные последовательности, а их операции (такие как ++, != и *) не должны генерировать исключений при работе с действительными итераторами.

Итераторы — это типичные примеры объектов, которые свободно копируются стандартными алгоритмами и операциями над стандартными контейнерами. Соответственно, копирующие операции итераторов не должны генерировать исключений, и стандарт гарантирует, что копирующие операции итераторов, возвращаемых стандартными контейнерами, исключений не генерируют. Например, итератор, возвращаемый посредством ***vector<T>::begin***(), можно копировать, не опасаясь исключений.

В то же время, операции ++ и -- для итераторов могут генерировать исключения. К примеру, ***istreambuf_iterator*** (§19.2.6) генерирует исключение для указания на ошибку ввода, а итератор с проверкой диапазона может сгенерировать исключение для указания на попытку выхода за границы действительного диапазона (§19.3). Но исключения не могут генерироваться при перемещении итератора от одного элемента последовательности к другому без нарушения определений операций ++ и -- для итераторов. Поэтому ***uninitialized_fill***(), ***uninitialized_fill_n***() и ***uninitialized_copy***() предполагают, что операции ++ и -- на их параметрах-итераторах исключений не порождают; если все же исключение генерируется, то это может быть следствием передачи под видом итераторов вовсе и не итераторов (в стандартном понимании), или что итерируемая последовательность не является действительной последовательностью. И вновь подчеркиваем, что стандартные контейнеры не защищают пользователя от порождаемого пользователем неопределенного поведения кода (§Е.2).

Е.4.5. Ссылки на элементы

Когда ссылка, указатель или итератор на элемент контейнера передается некоторому коду, этот код может испортить контейнер, повредив его элемент. Например:

```
void f(const X& x)
{
  list<X> lst;
  lst.push_back(x);
  list<X>::iterator i = lst.begin();
  *i = x;        // копируем x в list
  // ...
}
```

Если x испорчен, деструктор списка возможно будет не способен должным образом уничтожить *lst*. Например:

```
struct X
{
int* p;
 X() {p = new int; }
~X() {delete p; }
//...
};

void malicious()
{
 X x;
 x.p = reinterpret_cast<int*>(7);    // портим x
 f(x);                               // бомба с часовым механизмом
}
```

Когда исполнение функции *f*() завершается, вызывается деструктор типа *list*<*X*>, что, в свою очередь, порождает вызов деструктора типа *X* для испорченного значения. Но результат для ***delete p***, когда ***p*** отличен от нуля и не указывает на действительное значение типа *X*, не определен и может спровоцировать немедленный крах, или будет испорчена свободная память, так что проблемы проявятся неожиданным образом позже и в иных частях программы.

Возможность возникновения таких проблем не должна останавливать нас от манипулирования элементами контейнеров посредством ссылок или итераторов; это зачастую самый простой и наиболее эффективный способ. Однако будет разумным проявить дополнительную предосторожность в работе со ссылками на элементы контейнеров. Когда целостность контейнера имеет особое значение, будет неплохо предложить менее опытным программистам более безопасную альтернативу. Например, можно предоставить операцию, которая проверяет действительность нового элемента перед его копированием в особо важный контейнер. Естественно, осуществить такую проверку без детальных знаний прикладного типа невозможно.

В самом общем случае, порча элемента контейнера может способствовать непредсказуемому поведению последующих операций с контейнером. И это никакая не уникальная особенность контейнеров. Любой объект, оставленный в плохом состоянии, может впоследствии вызвать сбой программы.

Е.4.6. Предикаты

Многие стандартные алгоритмы и многие операции над стандартными контейнерами полагаются на пользовательские предикаты. В частности, все ассоциативные контейнеры нуждаются в предикатах как для поиска, так и для вставки элементов.

Предикат, используемый операцией стандартного контейнера, может генерировать исключения. На этот случай каждая операция стандартного контейнера предоставляет базовую гарантию, а некоторые операции, такие как ***insert***() для единственного элемента, обеспечивают сильную гарантию (§Е.4.1). Если предикат генерирует исключение во время операции над контейнером, окончательный набор элементов контейнера может оказаться не совсем таким, какой хотел пользователь,

но это будет набор действительных элементов. Например, если операция сравнения == сгенерирует исключение, будучи вызванной из ***list*::*unique*()** (§17.2.2.3), пользователь не сможет твердо рассчитывать на отсутствие дубликатов в списке. Он может рассчитывать лишь на то, что каждый элемент в списке является действительным (см. §Е.5.3).

По счастью, предикаты редко делают что-нибудь, что может генерировать исключение. Однако пользовательские предикаты <, == и != все же нужно принимать во внимание при рассмотрении вопросов, связанных с безопасностью кода при исключениях.

Объект сравнения ассоциативного контейнера копируется в процессе выполнения операции ***swap*()** (§Е.4.3). Поэтому имеет смысл позаботиться о том, чтобы копирующие операции предикатов, которые могут использоваться в качестве объектов сравнения, не генерировали бы исключений.

Е.5. Другие части стандартной библиотеки

Главная задача в контексте безопасности исключений — это поддержание согласованности объектов; то есть нужно не только поддерживать инварианты индивидуальных объектов, но и согласовывать их между собой. С точки зрения стандартной библиотеки наибольшую трудность в деле обеспечения безопасности исключений представляют собой объекты-контейнеры, а остальная часть библиотеки менее интересна в этом отношении. В то же время, и встроенные массивы являются контейнерами, которые могут быть испорчены какими-либо опасными операциями.

В общем случае, функции стандартной библиотеки генерируют только те исключения, которые включены в их спецификацию, плюс исключения, которые генерируются при этом вызываемыми пользовательскими операциями. Кроме того, все функции, которые прямо или косвенно выделяют память, могут генерировать исключение, свидетельствующее об исчерпании памяти (как правило, ***std*::*bad_alloc***).

Е.5.1. Строки

Операции над строками могут генерировать множество разных исключений. Однако ***basic_string*** манипулирует символами посредством функций, предоставляемых ***char_traits*** (§20.2), а этим функциям запрещено генерировать исключения. Подчеркнем, что именно функции ***char_traits*** из стандартной библиотеки не генерируют исключений, в то время как пользовательские ***char_traits*** могут и не соблюдать этого соглашения. Обратите также внимание на то, что типу, который используется в качестве символьного типа (типа элементов) для ***basic_string***, запрещено иметь определяемые пользователем операции копирования. Тем самым устраняется важный потенциальный источник исключений.

Тип ***basic_string*** во многих отношениях похож на стандартный контейнер (§17.5, §20.3). Его элементы фактически составляют последовательность, к которой можно обращаться с помощью ***basic_string*<*Ch*, *Tr*, *A*>::*iterator*** и ***basic_string*<*Ch*, *Tr*, *A*>::*const_iterator***. Соответственно, реализация строк стандартной библиотеки предоставляет базовую гарантию (§Е.2), а гарантии для ***erase*()**, ***insert*()**, ***push_back*()** и ***swap*()** (Е.4.1) распространяются и на ***basic_string***. Например, ***basic_string*<*Ch*, *Tr*, *A*>::*push_back*()** обеспечивает сильную гарантию.

Е.5.2. Потоки

Когда нужно, функции типа ***iostream*** генерируют исключения, чтобы сообщить об изменениях состояния (§21.3.6). Соответствующая семантика хорошо определена и не создает проблем по части безопасности исключений. Если же пользовательские ***operator<<<*** () или ***operator>>>*** () генерируют исключения, то это может выглядеть так, что исключения сгенерировала библиотека. Такие исключения, однако, не влияют на состояние потока (§21.3.3). Дальнейшие операции потока может быть и не найдут ожидаемых данных — из-за того, что предшествовавшая операция вместо нормального завершения сгенерировала исключение — но сам по себе поток не поврежден. Как всегда при возникновении проблем ввода/вывода, перед возобновлением чтения или записи может потребоваться вызов функции ***clear*** () (§21.3.3, §21.3.5).

Подобно ***basic_string***, потоки ***iostream*** в своих манипуляциях с символами полагаются на ***char_traits*** (§20.2.1, §Е.5.1). Таким образом, реализация может полагаться на то, что операции с символами не генерируют исключений, но не дается никаких гарантий в отношении нарушения этих соглашений пользователем.

Ради возможности серьезной оптимизации делается предположение, что классы ***locale*** (§D.2) и ***facet*** (§D.3) не генерируют исключений. Если бы они генерировали исключения, то использующий их поток мог бы быть испорчен. Наиболее вероятное исключение, ***std::bad_cast***, может генерироваться в рамках пользовательского кода вне реализации стандартных потоков. В самом худшем случае это может привести к неполному выводу или к неудачному чтению, но не к повреждению потока ***ostream*** (или ***istream***).

Е.5.3. Алгоритмы

Кроме ***uninitialized_fill*** (), ***uninitialized_fill_n*** () и ***uninitialized_copy*** () (§Е.4.4), по стандарту для алгоритмов предлагается лишь базовая гарантия (§Е.2). То есть в случае корректного поведения пользовательских объектов стандартные алгоритмы соблюдают все инварианты стандартной библиотеки и не порождают утечку ресурсов. Во избежание неопределенного поведения пользовательские операции всегда должны оставлять свои операнды в действительных состояниях, а деструкторы не должны генерировать исключений.

Сами алгоритмы не генерируют исключений. Вместо этого они сообщают об ошибках через свои возвращаемые значения. Например, поисковые алгоритмы в качестве сообщения «не найдено» обычно возвращают ссылку на конец последовательности (§18.2). Таким образом, исключения, сгенерированные в процессе работы стандартных алгоритмов, на самом деле генерируются пользовательскими операциями. Исключения эти возникают в таких операциях на элементах, как предикаты (§18.4), присваивания или ***swap*** (), а также в аллокаторах (§19.4).

Если перечисленные операции генерируют исключение, то алгоритм немедленно завершает работу, а дальнейшие решения по обработке исключения остаются за функциями, вызвавшими алгоритм. Некоторые алгоритмы допускают возникновение исключений в ситуациях, когда контейнер находится в недопустимом с точки зрения пользователя состоянии. Например, некоторые алгоритмы сортировки копируют элементы во временный буфер, а затем обратно помещают их в контейнер. Такой алгоритм ***sort*** () мог бы скопировать элементы из контейнера (планируя за-

писать их потом в контейнер в отсортированном порядке) и начать писать что-либо поверх них, а тут вот генерируется исключение. С точки зрения пользователя контейнер испорчен, но в то же время все элементы находятся в действительном состоянии, так что здесь возможно достаточно прямолинейное восстановление.

Обратим внимание на то, что стандартные алгоритмы обращаются к последовательностям через итераторы. Таким образом, стандартные алгоритмы не работают с контейнерами напрямую, а работают лишь с их элементами. Тот факт, что стандартные алгоритмы никогда непосредственно не добавляют и не удаляют элементы контейнеров, упрощает анализ последствий исключения. Аналогичным образом, если к некоторой структуре данных обращаются лишь через итераторы, указатели или ссылки на константы (например, с помощью *const Rec**), то легко доказать, что исключения в этом случае не приводят к нежелательным последствиям.

Е.5.4. Типы valarray и complex

Числовые функции не генерируют исключений явным образом (глава 22). В то же время, valarray должен распределять память, что может привести к исключению *std::bad_alloc*. Кроме того, шаблонам *valarray* и *complex* можно передать тип элементов (скалярный тип), который в общем случае генерирует исключения. Как всегда, стандартная библиотека предоставляет базовую гарантию (§Е.2), но никаких специальных гарантий относительно результатов вычислений, прерванных исключением, не дается.

Подобно *basic_string* (§Е.5.1) типы *valarray* и *complex* полагаются на то, что пользовательским типам их аргументов запрещено явным образом определять копирующие операции, так что их копирование побитовое. Как правило, эти числовые типы стандартной библиотеки оптимизированы по быстродействию в предположении, что типы их элементов (скалярные типы) не генерируют исключений.

Е.5.5. Стандартная библиотека языка С

Генерирует ли исключения операция стандартной библиотеки, не имеющая спецификации исключений, или не генерирует, зависит от конкретной реализации. Функции стандартной библиотеки языка С не уполномочены генерировать исключения по определению (в языке С нет исключений), но это могут делать их аргументы-функции. Реализации часто объявляют стандартные библиотечные функции языка С с пустой спецификацией исключений *throw*() с целью помочь компилятору в их оптимизации.

Функции типа *qsort*() и *bsearch*() (§18.11) принимают в качестве аргумента указатель на функцию. Поэтому они могут сгенерировать исключение, если это делают их аргументы. Базовая гарантия (§Е.2) распространяется и на эти функции.

Е.6. Рекомендации пользователям стандартной библиотеки

Можно глядеть на безопасность исключений в контексте стандартной библиотеки таким образом, что все проблемы мы создаем себе сами. Библиотека будет функционировать корректно до тех пор, пока предоставляемые пользователем операции

соответствуют базовым гарантиям библиотеки (§Е.2). В частности, исключения, сгенерированные из стандартных контейнерных операций не вызовут утечек памяти и не оставят контейнеры в недействительных состояниях. В итоге, перед пользователем библиотеки встает вопрос: каким образом определять свои типы, чтобы они не вызывали утечек ресурсов и не допускали неопределенного поведения?

Вот основные правила:

1. При модификации объекта не уничтожайте его старое содержимое до того, как новое содержимое полностью сконструировано и может заменить старое содержимое без риска возникновения исключений. Например, см. реализацию ***vector*::*operator*=**(), ***safe_assign***() и ***vector*::*push_back***() в §Е.3.
2. Перед генерацией исключения освободите захваченные ресурсы, которые не принадлежат какому-либо другому объекту.
   - **2a.** Методика «выделение ресурса есть инициализация» (§14.4) и языковые правила, гласящие, что частично созданные объекты уничтожаются в той степени, в которой они были созданы (§14.4.1), оказывают здесь большую помощь. К примеру, см. ***leak***() в §Е.2.
   - **2b.** Алгоритм ***uninitialized_copy***() и его собратья обеспечивают автоматическое освобождение ресурсов в случае невозможности завершить конструирование набора объектов (§Е.4.4).
3. Перед генерацией исключения удостоверьтесь, что все операнды находятся в действительных состояниях. Всегда оставляйте объекты в состоянии, которое позволяет обратиться к ним и уничтожить их, не вызвав неопределенного поведения или генерации исключения из деструктора. К примеру, см. присваивание для ***vector*** из §Е.3.2.
   - **3a.** Конструкторы имеют такую особенность, что при генерации в них исключений не остается объектов для последующего уничтожения. Соответственно, перед генерацией исключения нам нет необходимости устанавливать инвариант, но есть необходимость в освобождении ресурсов, захваченных в ходе неудавшегося конструирования.
   - **3b.** Деструкторы особенны тем, что сгенерированное в них исключение почти наверняка ведет к нарушению инвариантов и/или вызову ***terminate***().

Следовать этим правилам на практике не так-то легко. Главная причина состоит в том, что исключения иногда появляются оттуда, откуда их совсем не ждут. Хороший пример — ***std*::*bad_alloc***. Любая функция, которая прямо или косвенно использует операцию ***new*** или обращается к аллокатору для выделения памяти, может сгенерировать ***bad_alloc***. В некоторых программах мы можем решить эту частную проблему, стараясь не исчерпать память компьютера. Однако для программ, которые должны работать в течение длительных промежутков времени или должны осуществлять ввод данных произвольного объема, нужно быть готовым к обработке самых разнообразных отказов при выделении принципиально ограниченных компьютерных ресурсов. Каждую функцию нужно подозревать на предмет возможности генерации исключений до тех пор, пока твердо не доказано обратное.

Простой способ попробовать избежать неожиданностей состоит в применении контейнеров элементов, которые не генерируют исключений (вроде контейнеров

указателей или контейнеров элементов простых конкретных типов), или связных контейнеров типа ***list,*** предоставляющих сильную гарантию (§Е.4). Другой подход — положиться в первую очередь на операции вроде ***push_back*** (), которые предоставляют сильную гарантию, что операция либо успешно заканчивается, либо ни на что не влияет (§Е.2). Однако сами по себе эти подходы недостаточны для предотвращения утечек ресурсов и могут приводить к слишком ограничительным и пессимистическим методам обработки ошибок и восстановления. Например, ***vector<T*>*** — тривиально безопасный по отношению к исключениям тип в том случае, если операции над типом ***T*** не генерируют исключений. Однако если указуемые объекты нигде не удаляются, исключение из ***vector*** приведет к утечке ресурсов. Введение вспомогательного класса ***Handle*** (§25.7), предназначенного для освобождения ресурсов, и применение ***vector<Handle<T> >*** вместо ***vector<T*>***, с большой вероятностью улучшит жизнеспособность кода.

Если вы пишите новый код, то можно с самого начала принять систематический план, по которому каждый ресурс представляется классом с инвариантом, предоставляющим базовую гарантию (§Е.2). Таким образом идентифицируются наиболее важные объекты системы, для которых можно обеспечить *семантику отката* (*roll-back semantics*), то есть сильную гарантию для операций с ними (возможно, при каких-то специфических условиях).

Большинство приложений содержат структуры данных и код, которые написаны без учета безопасности исключений. При необходимости такой код можно встраивать в более надежный (с точки зрения безопасности исключений) каркас либо после проверки, что он не генерирует исключений (как это было для функций стандартной библиотеки языка С; §Е.5.5), либо посредством интерфейсных классов, для которых поведение при исключениях и управление ресурсами может быть точно определено.

Проектируя типы, предназначенные для использования в безопасной по отношению к исключениям среде, нужно обращать особое внимание на операции, используемые стандартной библиотекой: конструкторы, деструкторы, присваивания, сравнения, функции ***swap*** (), предикаты и операции на итераторах. Наилучшее решение состоит в определении инварианта класса, который устанавливается конструкторами. Иногда нужно проектировать инвариант класса так, чтобы объект можно было поместить в такое состояние, при котором его можно уничтожить, даже если операция потерпела неудачу в самой «неудобной» точке. В идеале, такое состояние не должно быть артефактом реализации, искусственно введенным ради удобства обработки исключений, а должно естественным образом вытекать из семантики типа (§Е.3.5).

Рассматривая безопасность исключений, делайте акцент на определении действительных состояний объектов (инвариантов) и на надлежащем освобождении ресурсов. Из-за этого тем более важно представлять ресурсы классами. Простым примером служит ***vector_base*** (§Е.3.2). Конструкторы таких «ресурсных» классов захватывают низкоуровневые ресурсы (типа «сырой» памяти для ***vector_base***) и устанавливают инварианты (вроде надлежащей инициализации указателей в ***vector_base***). Деструкторы этих классов неявно освобождают низкоуровневые ресурсы. Правила частичного конструирования (§14.4.1) и методика «выделение ресурса есть инициализация» (§14.4) поддерживают такой способ управления ресурсами.

Хорошо написанный конструктор устанавливает инвариант класса для объекта (§24.3.7.1), то есть он присваивает объекту такое значение, которое способствует упрощению написания кода для операций класса и способствует их успешному выполнению. Это подразумевает, что конструкторам часто приходится захватывать ресурсы. Если же это не удается сделать, то конструкторы могут сгенерировать исключение, чтобы мы попытались справиться с проблемой до того, как объект создан. Такой подход непосредственно поддерживается и языком, и стандартной библиотекой (§E.3.5).

Требование освободить ресурсы и перевести объект в действительное состояние до того момента, когда будет генерироваться исключение, означает, что бремя обработки исключения распределяется между генерирующей исключение функцией, функциями, расположенными по цепочке вызовов между последней и обработчиком, и самим обработчиком. Таким образом, если функция генерирует исключение, это не означает, что она никак не связана с его обработкой: все функции, генерирующие и передающие исключение должны освобождать захваченные ими ресурсы и помещать операнды в согласованные состояния. Если они этого не делают, то обработчику исключений скорее всего остается лишь попытаться более-менее изящно завершить работу приложения.

E.7. Советы

1. Определитесь с тем, какая степень безопасности исключений нужна вам; §E.2.
2. Безопасность исключений должна быть составной частью общей стратегии отказоустойчивости; §E.2.
3. Предоставляйте базовую гарантию для всех классов (то есть поддерживайте инвариант и не допускайте утечек ресурсов); §E.2, §E.3.2, §E.4.
4. Где это возможно и позволительно, предоставляйте сильную гарантию (то есть чтобы операция либо успешно выполнялась, либо оставляла операнды нетронутыми); §E.2, §E.3.
5. Не генерируйте исключений в деструкторах; §E.2, §E.3.2, §E.4.
6. Не генерируйте исключений в итераторах, которые осуществляют проход по действительной последовательности; §E.4.1, §E.4.4.
7. Безопасность исключений требует тщательного изучения отдельных операций; §E.3.
8. Проектируйте шаблоны так, чтобы они были прозрачными по отношению к исключениям; §E.3.1.
9. В вопросе приобретения ресурсов отдавайте предпочтение конструкторам, а не функциям ***init***(); §E.3.5.
10. Определяйте инвариант класса так, чтобы было ясно, какое состояние объектов класса является действительным; §E.2, §E.6.
11. Убедитесь, что объект всегда можно поместить в действительное состояние без риска генерации исключения; §E.3.2, §E.6.

12. Придерживайтесь простых инвариантов; §Е.3.5.
13. Перед генерацией исключения оставляйте операнды в действительных состояниях; §Е.2, §Е.6.
14. Избегайте утечек ресурсов; §Е.2, §Е.3.1, §Е.6.
15. Представляйте ресурсы непосредственно классами; §Е.3.2, §Е.6.
16. Помните, что ***swap***() может быть альтернативой копированию элементов; §Е.3.3.
17. Там, где возможно, предпочитайте упорядочение кода явному применению ***try***-блоков; §Е.3.4.
18. Не уничтожайте «старую» информацию до того, как заменяющая ее информация будет надежно подготовлена; §Е.3.3, §Е.6.
19. Полагайтесь на методику «выделение ресурса есть инициализация»; §Е.3, §Е.3.2, §Е.6.
20. Удостоверьтесь, что объекты сравнения ассоциативных контейнеров можно копировать; §Е.3.3.
21. Выделяйте наиболее важные структуры данных и предоставляйте сильную гарантию для их операций; §Е.6.

Е.8. Упражнения

1. (*1) Перечислите все исключения, которые могут быть сгенерированы в функции ***f***() из §Е.1.
2. (*1) Ответьте на вопросы к примеру из §Е.1.
3. (*1) Определите класс ***Tester***, который иногда генерирует исключения в таких базовых операциях, как копирующий конструктор. С помощью ***Tester*** проверьте контейнеры вашей реализации стандартной библиотеки.
4. (*1) Найдите ошибку в «неряшливой» версии конструктора ***vector*** (§Е.3.1) и напишите программу, приводящую этот конструктор к краху. Подсказка: сначала реализуйте деструктор для типа ***vector***.
5. (*2) Реализуйте простой список, предоставляющий базовую гарантию. Уточните, что список требует от пользователей, чтобы обеспечить указанную гарантию.
6. (*3) Реализуйте простой список, предоставляющий сильную гарантию. Тщательно протестируйте этот список. Как люди могут убедиться в безопасности этого списка?
7. (*2.5) Реализуйте ***String*** из §11.12 так, чтобы он стал столь же безопасным, как стандартный контейнер.
8. (*2) Сравните производительность различных версий присваивания для ***vector*** и ***safe_assign***() (§Е.3.3).
9. (*1.5) Скопируйте аллокатор без использования операции присваивания (как это требуется для улучшения ***operator=***() в §Е.3.3).

10. (*2) Добавьте к типу ***vector*** одноэлементные и многоэлементные операции ***erase***() и ***insert***(), предоставляющие базовую гарантию (§Е.3.2).
11. (*2) Добавьте к типу ***vector*** одноэлементные и многоэлементные операции ***erase***() и ***insert***(), предоставляющие сильную гарантию (§Е.3.2). Сравните стоимость и сложность данного решения с решением из упражнения 10.
12. (*2) Напишите ***safe_insert***() (§Е.4.2), которая вставляет элементы в существующий ***vector*** (а не копирует во временную переменную). Какие ограничения придется наложить на операции?
13. (*2) Напишите ***safe_insert***() (§Е.4.2), которая вставляет элементы в существующий ***map*** (а не копирует во временную переменную). Какие ограничения придется наложить на операции?
14. (*2.5) Сравните размер, сложность и производительность ***safe_insert***() из упражнений 12 и 13 с версией ***safe_insert***() из §Е.4.2.
15. (*2.5) Напишите усовершенствованный вариант ***safe_insert***() исключительно для ассоциативных контейнеров. Используйте ***traits*** для написания ***safe_insert***(), который автоматически выбирает оптимальный вариант для контейнера. Подсказка: §19.2.3.
16. (*2.5) Попробуйте переписать ***uninitialized_fill***() (§19.4.4, §Е.3.1) так, чтобы он справлялся с деструкторами, генерирующими исключения. Возможно ли это? Если да, то какова стоимость такого решения? Если нет, то почему?
17. (*2.5) Попробуйте переписать ***uninitialized_fill***() (§19.4.4, §Е.3.1) так, чтобы он справлялся с итераторами, которые генерируют исключения в операциях - и ++. Возможно ли это? Если да, то какова стоимость такого решения? Если нет, то почему?
18. (*3) Выберите контейнер из библиотеки, отличной от стандартной. Ознакомьтесь с документацией и выясните, какие гарантии безопасности исключений он предоставляет. Испытайте этот контейнер, чтобы выяснить, насколько он устойчив к исключениям, генерируемым при выделении памяти, а также кодом, предоставляемым пользователем. Сравните с соответствующим контейнером стандартной библиотеки.
19. (*3) Попробуйте оптимизировать ***vector*** из §Е.3, пренебрегая возможностью исключений. Например, устраните все ***try***-блоки. Сравните производительность с версией из §Е.3 и с реализацией ***vector*** из стандартной библиотеки. Сравните также размер и сложность кода этих вариантов типа ***vector***.
20. (*1) Определите инварианты для ***vector*** (§Е.3) с возможностью ***v==0*** и без таковой (§Е.3.5).
21. (*2.5) Прочитайте исходный код реализации vector. Какие гарантии обеспечены для присваивания, многоэлементных ***insert***() и ***resize***()?
22. (*3) Напишите версию ***hash_map*** (§17.6), которая столь же безопасна, что и стандартный контейнер.

Предметный указатель

!

[], 65, 557
>>, 158, 174
#define, 274
#endif, 274
#ifndef, 274
%d, 207
&, 158, 174
*, 65
*/, 78
*this, 292
, (запятая), 173
/*, 78
^, 158, 174
__ioinit, 754
|, 158, 174
~, 174
~container(), 558
~ios_base(), 766
>>, 158, 174

A

abort(), 276
abstract node class, 905
– types, 74
access functions, 531
accumulate(), 801
acos(), 779
adapters, 565, 620
adjacent_difference(), 801
–_find(), 609
alert, 964
algorithm, 521, 800
aliases, 234
allocate(n), 675
allocator_type, 556
allocators, 674
and, 926
–_eq, 926
ANSI, 46
appending, 703
application frameworks, 918, 919
argc, 167
argv, 167
array, 559
ASCII, 963
–-символы, 690
asin(), 779
asm, 926
assertion, 879
assign(first,last), 558
– (n,x), 558
assignment, 309
– operator, 290
associative array, 97, 576
– containers, 555
at(), 557
atof(), 712
atoi(), 712
atol(), 712
auto, 926
automatic objects, 129

B

back(), 557, 573, 695
–_inserter(), 99, 662
backslash, 117, 964
backspace, 964
backward_copy(), 633
bad(), 729
–_alloc, 467
–_cast, 467
–_exception, 467
–_typeid, 467
base class, 373
basefield, 728
Basic Linear Algebra Subprograms, 786
–_ios, 735, 759
–_ostringstream, 755
–_streambuf, 764
–_string, 691, 693
–_stringCh, 694
–_stringchar, 693
begin(), 557
BEL, 137

binary_function, 620
–_negate, 621
–_search(), 611
bind1st, 620
–(), 622
–(x), 620
bind2nd, 620
–(), 622
–(y), 620
binder1st, 622
bitand, 926
bitor, 926
bitset, 520, 555, 559
bitwise logical operators, 174
bool, 62, 114, 171, 926
break, 926
built-in types, 114

C

caching, 293
callback function, 388
capacity(), 558, 565
carriage return, 964
case, 926
–-ветви, 64
–-метки, 158
cassert, 521
casting, 181
catch, 927
–(...), 442
–-блок, 69, 437
cctype, 521-522
cerr, 165
cerrno, 521, 779
cfloat, 522
char, 62, 64, 114, 171, 927
–*, 65
–_BIT, 778
–_traits, 720, 734
–_traitschar, 691
–_traitswchar_t, 692
character constant, 88
characters, 690
cin, 63, 90, 168, 728
class, 151, 286, 927
– declaration, 285
– definition, 285
– hierarchy, 378, 473
classes inheritance, 373
cleanup, 73
clear(), 557, 707
climits, 522
clocale, 522
cmath, 523, 779
codecvt<Ch,char,mbstate_t>, 1027
collate_byname<Ch>, 1027
collate<Ch>, 1027
comma operator, 173
compare(), 692
compile-time polymorphism, 424
compl, 927
complex, 523
concatenation, 88, 704
concrete node class, 905
– types, 74, 303
conditional-expressions, 186
const, 50, 256, 927
–_cast, 927
–_iterator, 556
–_mem_fun_ref_t, 621
–_mem_fun_t, 620
–_mem_fun1_ref_t, 621
–_mem_fun1_t, 620
–_reference, 556
–_reverse_iterator, 556
constant expressions, 967
constructors, 287
container, 890
– adapter, 571
– classes, 82, 379
–(), 558
–(first,last), 558
–(n), 558
–(n,x), 558
–(x), 558
continue, 926
conversion operators, 343
copy constructor, 290, 309
–(), 82, 106, 610, 633
–_backward(), 610
copyfmt(), 766
copy-on-write, 365
cos(), 108
count (), 106
count(), 609
count_if(), 106, 609
cout, 63
crosscast, 495
csetjmp, 522
cshift(), 784
csignal, 522
cstdarg, 522
cstddef, 522
cstdio, 259, 522, 733
cstdlib, 277, 521-523, 779

cstring, 521
ctime, 520, 522
Ctrl+Z, 166
ctype_byname<Ch>, 1027
ctype<Ch>, 1027
curr_symbol(), 1041
cwchar, 522, 710
cwcharp, 521
cwctype, 521, 713
C-строки, 90, 711

D

deallocate(p,n), 675
decimal, 118
—_point(), 1041
declarations, 124, 930
declarator, 125
default, 926
—-ветвь, 64, 91, 158
definitions, 124
delegation, 360, 875
delete, 180, 926
deque, 520, 556, 559, 565
dereference operator, 83
dereferencing, 134
design reuse, 833
destructive copy semantics, 448
destructor, 73, 305
diamond-shaped inheritance, 484
dictionary, 97, 576
difference_type, 556
digraphs, 690
divides, 619
do, 926
—_get_time(), 1053
dot operator, 149
— product, 803
double, 62, 114, 119, 926
— quote, 964
—-ended queue, 570
—-linked list, 565
downcast, 494
dynamic memory, 979
— store, 74
—_cast, 926

E

ellipsis, 202, 207
else, 926
empty string, 137
—(), 558
end(), 557
enum, 50, 114, 151, 926
enumeration, 122
EOF, 712, 734, 769
eofbit, 175
eq(), 692
equal(), 609
—_range(), 106, 587, 611
—_range(k), 559
—_to, 618
erase(), 96, 548, 599, 706-707
—(first,last), 557
—(p), 557
errno, 779
escape-символы, 964
exception, 242, 467, 521-522
— handler, 243
— safety, 1078
— specification, 457
—(), 735
exit(), 276
explicit, 926
export, 927
expression, 930
—-oriented coding, 176
extern, 927
external linkage, 255
extractor, 720

F

facets, 1007
factory, 396
fail(), 729
false, 64, 115, 927
fat interface, 529, 891
fill(), 610
fill_n(), 610
find(), 106, 609, 627-628
find(k), 559
find_end(), 609
—_first_of, 609
—_if(), 106, 609, 628
float, 113, 119, 927
floating-point literal, 119
flush(), 746
for, 927
—(), 158
—_each (), 106, 609
form feed, 964
format string, 207
friend, 927
front(), 557, 573, 695
front_inserter(), 662

fstream, 522, 760
full expression, 318
function declaration, 195
— definition, 196
— objects, 356, 607
functional, 520
function-like objects, 616
—-style cast, 183
functors, 356, 616

G—H

garbage collection, 308
generalized slice, 796
generate(), 610
generate_n(), 610
generic programming, 401
get(), 728,732
get_allocator(), 558
—_temporary_buffer(), 683
getchar(), 733
getline(), 709, 732
gmtime(), 1048
good(), 729
goto, 926
gptr(), 761
greater, 618
—_equal, 618
grouping(), 1041
gslice_array, 780, 786
handle, 352, 363, 914
handles, 895, 914-915, 917
has-a relationship, 869
header file, 67
heap, 74, 648, 979
— objects, 129
helper functions, 302, 341, 533, 550
hex number, 964
hexadecimal, 118
horizontal tab, 964

I—K

if, 926
imbue(), 766
—(loc), 763
implementation inheritance, 871
implicit conversion, 121
includes(), 611
increment operator, 83
incremental changes, 803
indirect_array, 780, 786
indirection, 134, 360
inheritance, 80
initializer, 125
inline, 50, 256, 926
— functions, 196
inner_product(), 801
inplace_merge(), 611
input iterators, 102
insert(), 548
insert(p,first,last), 557
—(p,n,x), 557
—(p,x), 557
inserter, 720
—(), 662
int, 62, 113-114, 171, 926
—_MAX, 778
integral promotion, 968
interface, 75
— inheritance, 871
internal linkage, 256
internationalization, 1008
invalid_argument, 467
invariant, 877
iomanip, 522, 724, 747-748
ios, 467, 521, 747
—_base, 759
—_base::badbit, 759
—_base::failbit, 759
—_base::failure, 467
iosfwd, 521
iostream, 521, 709, 726
isalnum(), 164
isalpha(), 164
isdigit(), 164
ISO C++ (ISO/IEC 14882), 47
—/IEC 14882, 947
istream, 522, 729, 747, 760
—_iterator, 102, 666
istream
istringstream, 167, 756
iter_swap(), 610
iterator, 82, 520, 556
key, 576
—_comp(), 559, 582
—_compare, 556
—_type, 556

L

labels, 189
lazy evaluation, 295
leaf class, 905
left value, 130

left-associative, 171
less, 618
–_equal, 618
lexicographical_compare(), 612, 701
limits, 522
linkage, 254-255, 257, 259, 261, 263
– block, 263
– convention, 262
linker, 254
linking, 930
Liskov substitution principle, 871
list, 85, 87, 95, 97, 520, 556, 559, 565
loader, 254
locale, 522
localization, 1008
localtime(), 1048
log(), 108
logical constness, 292
–_and, 618
–_not, 618
–_or, 618
long, 926
– double, 119
lower_boud(k), 559
–_bound(()), 611
–_bound(), 587
lt(), 692
lvalue, 130

M

make_heap(), 612
malloc(), 50
manipulators, 741
map, 85, 97, 520, 556, 559, 576
mapped value, 576
–_type, 556
mask_array, 780, 786
max(), 612
–_element(), 612
–_size(), 558
mem_fun(), 620
–_fun_ref(), 621
–_fun_ref_t, 621
–_fun_t, 620
–_fun1_ref_t, 621
–_fun1_t, 620
member functions, 285
– initializer list, 310
memory, 520
– leak, 310
– on the stack, 979
merge(), 106, 566-567, 611
messages_byname<Ch>, 1027
messages<Ch>, 1027
min(), 612
–_element(), 612
minus, 619
mismatch(), 609
module, 65
modulus, 619
money_get<C,In>, 1027
–_get<Ch>, 1027
–_put<C,Out>, 1027
–_put<Ch>, 1027
moneypunct<Ch>, 1027
–<Ch,International>, 1027
–<Ch,true>, 1027
multimap, 559
multiple inheritance, 392, 474
multiplies, 619
multiset, 559
mutable, 926

N

namespace, 50, 66, 926
narrow(ch,def), 1065
negate, 619
negative_sign(), 1041
nested class, 347, 362
new, 180, 467, 522, 926
– [], 180
– (buf) X, 320
–_handler(), 675
newline, 964
next_permutation(), 612, 650
node class, 903
not, 926
–_eq, 927
–_equal_to, 618
not1(), 621
not2(), 621
nth_element(), 611
null element, 656
– pointer, 134
num_get<C,In>, 1027
–_get<Ch>, 1027
–_put<C,Out>, 1027
–_put<Ch>, 1027
numeric, 523
–_limits, 777
numpunct_byname<Ch>, 1027
numpunct<Ch>, 1027

O

object hierarchies, 868
octal, 118
– number, 964
operator, 927
–[] (k), 559
– function, 329
– new [] (), 180
– –(), 180
–=(x), 558
or, 927
–_eq, 927
ort_heap(), 648
ostream, 522, 720, 747, 729, 757, 760
–_iterator, 102, 665
ostringstream, 755
out_of_range, 467
overflow_error, 467
overloading, 201
ownership semantics, 448

P–Q

pair<const Key,T>, 581
parametric polymorphism, 424
partial specialization, 418
–_sort(), 611
–_sort_copy(), 611
–_sum(), 801
partition(), 611
patterns, 833
permutations, 650
placement new, 320
plus, 619
point of instantiation, 409
pointer to member, 507
–_to_binary_function, 621
–_to_unary_function, 621
polymorphic type, 75, 383
polymorphism, 383
pop(), 65-66, 572
–_back(), 542, 548, 557
–_front(), 557, 573
–_heap(), 612, 648
positive_sign(), 1041
postfix, 125
pow(), 108
precedence, 171
precompiling, 258
prefix, 125
prev_permutation(), 612, 650
priority_queue, 559, 565, 574
private, 72, 927
promotions, 968
protected, 927
ptr_fun(), 621
pubimbue(), 759
–(loc), 763
public, 926
push(), 65-66, 573
–_back(), 542, 548, 557, 572-573
–_front(), 557
–_heap(), 612, 648
question mark, 964
queue, 520, 556, 559, 565, 572

R

random_shuffle(), 610
–-access iterators, 612
range, 613
raw memory, 181, 501, 676
rbegin(), 557
read(), 732
realloc(), 50
recursive descent, 156
reference, 144, 556
– count, 362
register, 926
reinterpret_cast, 601, 926
remove(), 610
remove_copy(), 610
–_copy_if(), 610
–_if(), 610
rend(), 557
replace(), 106, 610, 707
–_copy(()), 610
–_copy_if(), 610
–_if(), 610
representation, 914
– class, 363
– type, 352
reptace_if(), 106
–(), 558, 565
–(res_arg), 709
resize(), 558, 599
resolution rules, 202
return, 926
reverse(), 610
–_copy(), 610
–_iterator, 556, 663
right-associative, 171
rotate(), 610
–_copy(), 610

run-time errors, 244
– polymorphism, 424
– Type Information, 493, 495, 497, 499, 501, 503

S

scope, 127
– resolution operator, 128
search(), 82, 609
–_n(), 609
sentinel element, 83
sentries, 738
sentry, 738
separate compilation, 67
sequences, 82, 555
– of elements, 98
sequencing operator, 135
set, 520, 559, 588
–_difference(), 611
–_intersection(), 611
–_load(), 599
–_symmetric_difference(), 611
–_terminate(), 462
–_union(), 611
setstate(failbit), 731
shift state, 1067
–(), 784
short, 926
–-circuit evaluation, 173
signed, 926
single inheritance, 474
– quote, 964
size(), 558
–_type, 556
sizeof, 926
slice, 786
–_array, 780,786
smart pointers, 358
sort(), 82, 106, 566, 611
sort_heap(), 612
source file, 254, 930
specialization pattern, 418
specializations, 406
specifiers, 125
splice(), 566-567
sstream, 522,755
stable_partition(), 611
stable_sort(), 611
stack, 520, 556, 559, 565
stack unwinding, 436, 446
Standard Template Library, 109
state, 729
static, 926
– member, 289
– objects, 129
–_cast, 926
std::set_terminate(), 462
–_unexpected(), 462
stdexcept, 467, 521
stdio.h, 259
STL, 109
streambuf, 521, 751, 756-757, 759, 761
streamoff, 722
strict weak ordering, 562
stride, 787
string, 50, 85, 87, 521, 555, 559
– literal, 86, 136
stringstream, 755, 760
strtol(p,end,b), 712
struct, 286, 927
structure pointer dereference operator, 149
subclass, 373
subtyping, 871
superclass, 373
swap(), 558, 610
–_ranges(), 610
switch, 927
sync_with_stdio(false), 767

T

template, 927
– instantiation, 406
template<class T> , 81
temporary objects, 318
the one-definition rule, 260
– Standard Template Library, 56
this, 927
thousands_sep(), 1041
throw, 927
time(), 1048
–_get<Ch>, 1027
–_put_byname<Ch,Out>, 1027
–_put<Ch>, 1027
to_char_type(), 691
–_int_type(), 691
top(), 572, 574
traits_type::eof(), 733
–_type::to_int_type(), 761
transform(), 610
translation unit, 254
–-unit, 930
true, 115, 927
try, 926
–-блок, 69, 95, 244-245, 454

type field, 78
typedef, 256, 926
typeid, 926
typeinfo, 467, 522
typename, 926

U

uflow(), 761
unary_function, 620
—_negate, 621
underflow(), 761
Unicode, 56, 692
union, 151, 926, 977
unique(), 610
—_copy(), 106, 610
UNIX, 56
unnamed namespace, 232
unsetf(ios_base::skipws), 746
unsigned, 118,926
— char, 712
— int, 118
upcast, 495
uper_bound(()), 611
upper_bound(), 587
—_bound(k), 559
user specializations, 417
—-defined specializations, 417
—-defined types, 71
using, 926
—-declaration, 224
—-directive, 226
—-объявление, 477
utility, 520

V—W—X

valarray, 107, 136, 523, 555, 559
valid iterator, 656
value_comp(), 559, 582
—_type, 556
— types, 303
ve.begin(), 98
ve.end(), 98
vector, 50, 85, 97, 136, 520, 556, 559, 565
vertical tab, 964
virtual, 75, 125, 926
— base class, 481
void, 114, 926
—*, 50
volatile, 927
vtbl, 77
wchar.h, 710
wchar_t, 927
wctype.h, 713
while, 927
widen(), 1065
xor, 927
—_eq, 927

А

абстрактные классы, 80, 384-385, 390
— типы, 74, 895, 900-901
— — данных, 71
абстрактный узловой класс, 905
абстракция данных, 59, 69, 71, 73, 75, 77
автоматическая сборка мусора, 308
автоматический объект, 129
агрегат, 149
агрегация, 867
— и наследование, 869
адаптер queue, 572
— stack, 571
адаптеры, 565, 620
— последовательных контейнеров, 570-571, 573, 575
алгоритм accumulate(), 801
— adjacent_difference(), 803
— binary_search(), 645
— count(), 100, 629, 658
— count_if(), 105, 629
— equal(), 630
— fill(), 640
— find(), 100,104
— find_end(), 632
— find_if(), 105
— for_each, 103
— for_each(), 626, 635
— generate(), 640
— includes(), 647
— inner_product(), 802
— iter_swap(), 642
— lexicographical_compare(), 649
— merge(), 645
— mismatch(), 617, 630
— partial_sum(), 803
— partition(), 646
— remove(), 640
— remove_copy_if(), 640
— reverse(), 641
— rotate(), 641
— rotate_copy(), 641
— search(), 631
— search_n(), 632
— set_difference(), 647
— set_symmetric_difference(), 647

алгоритм stable_partition(), 646
– swap(), 642
– swap_ranges(), 642
– transform(), 634
– unique(), 636
– unique_copy(), 636
алгоритмы, 59, 98-99, 101, 103, 105, 1109
– partial_sort(), 643
– замены элементов, 638
– поиска, 627
– семейства partition, 646
– стандартной библиотеки, 608-609, 611
– стандартной библиотеки, 106
– удаления элементов, 640
– функциональный объектов, 607-652
аллокаторы, 655-686
альтернатива «агрегация/наследование», 873
– «включение/наследование», 871
анонимные пространства имен, 257
аргументы командной строки, 167
– по умолчанию, 206
– функциональных шаблонов, 410
арифметика указателей, 50
арифметическая операция -, 62
– – %, 62
– – *, 62
– – /, 62
– – +, 62
арифметические операции, 62-63
– типы, 114
ассоциативные контейнеры, 555, 576-587
– массивы, 45
ассоциативный контейнер multimap, 587
– – multiset, 588
– – set, 588
– массив, 97, 355, 576
– – map, 576

Б

базовый алгоритм sort(), 643
– класс, 373
– – basic_ios, 721, 726
базы данных определений типов, 42
беззнаковые целые типы, 118
беззнаковый интегральный тип size_t, 172
безопасности в контексте исключений, 1078
бесконечный цикл, 158, 188
библиотека BLAS, 786
библиотечная функция exit(), 165
– – printf(), 90
– – realloc(), 99
– – strlen(), 139
библиотечные предикаты, 618
библиотечный класс map, 169
– – ostream, 169
– – string, 169
– тип ptrdiff_t, 658
бинарная операция, 171, 330
бинарные предикаты, 617
бинарный поиск, 644
– предикат, 612
битовые поля, 976
– – bitset, 589
блок catch, 437
– спецификации компоновки, 263
буфера потоков, 666

В

ввод дат и времени, 1052
– и вывод дат и времени, 1046
– – финансовой информации, 1039
вектор, 45, 97
векторы, 971
виртуальные базовые классы, 480
– конструкторы, 396, 511
виртуальные функции, 59, 77, 381
виртуальный базовый класс, 481
вложенные пространства имен, 984
вложенный класс, 347,362
внешняя компоновка, 255
внутренняя компоновка, 256
возвращаемое значение, 200
восстановление типа исключения, 461
временные объекты, 318
вспомогательные функции, 533, 550
встроенные типы, 62, 114
встроенный тип, 42, 87
входные диапазоны, 614
– итераторы, 102
вывод денежных величин, 1043
выделение памяти под массивы, 511
вызов widen(c), 1065
выражение, 62
выражения, 169, 930
– и операторы, 155-194
вычитание указателей, 140

Г–Д

гарантии операций контейнеров, 1101
генерация исключений, 95
гибридное проектирование, 843
глобальная переменная errno, 434
графический сервер X-System, 48

двойная наклонная черта, 62
двойные кавычки, 136
двунаправленный итератор, 565, 612
двусвязный список, 97, 565
двусторонняя очередь, 97, 570
действительный итератор, 656
декларатор, 125, 128
— *const, 143
деклараторы, 938
декремент, 175, 360-361
делегирования, 875
дескриптор, 352, 363, 914
дескрипторные классы, 895, 914-915, 917
деструктор, 73-74, 305, 982
диапазоны, 613
диграфы, 690
динамическая память, 74, 684, 979
директива #endif, 215
— #ifdef, 216
— #include cmath, 259
— extern "C", 262-263
— компилятора #ifdef identifier, 215
директивы #include, 254, 257
— — препроцессора, 166
— using, 204, 226
— препроцессора, 944-945
— условной компиляции, 274
доступ к базовым классам, 491, 986
— к членам класса, 985
— к элементам массивов, 139
— ко вложенным классам, 988
дружественный класс, 348
друзья, 991
— класса, 346-347, 349

Е–Ж–З

единица трансляции, 254, 930
жирный интерфейс, 529, 890-891
зависимости внутри иерархии классов, 865
— типа "is-a", 870
заголовочные файлы, 166, 954
— — стандартной библиотеки, 259
заголовочный файл, 67
— — algorithm, 609
— — assert.h, 879
— — cassert, 879
— — cctype, 712, 734
— — climits, 778
— — clocale, 766
— — cmath, 778, 779
— — cstddef, 172, 658
— — cstdlib, 650, 711
— — ctype.h, 712
— — fstream, 752
— — functional, 609, 616, 618
— — iostream, 87, 734
— — istream, 726
— — limits.h, 778
— — limits, 776
— — list, 98
— — locale, 765, 1011, 1026
— — locale.h, 766
— — map, 98
— — math.h, 778-779
— — memory, 674
— — new, 180, 320, 684
— — numeric, 612, 800
— — queue, 572
— — sstream, 168
— — stdlib.h, 650, 711
— — streambuf, 756
— — string, 692, 709
— — utility, 578
— — valarray, 780
— — vector, 98
загрузчик, 254
закрытие потоков, 753
замаскированные указатели, 980
замещение функций виртуальных базовых классов, 486
защита от повторных включений, 274
защищенные члены классов, 489
знак *, 134
— =, 63
— ==, 63
— >>, 90
знаковые и беззнаковые символы, 966
знаковый интегральный тип ptrdiff_t, 172
значение EDOM, 779
— ERANGE, 779
— true, 105

И

идентификатор, 113, 126
иерархии классов, 80, 378, 473-514, 862
иерархия объектов, 868
имена, 126
именованные фасеты сравнения, 1032
имя, 62, 113, 126
инвариант, 877
инварианты, 877
инициализатор, 125
инициализаторы в операторах for, 958
— членов класса, 311
инициализация, 129

инициализация массивов, 135
– ссылок, 145
инициализирующие выражение, 128
инкапсуляция, 883
инкремент, 175, 360-361
интегральное продвижение, 968
интегральные типы, 114
интеллектуальные указатели, 358
интернационализация, 1008
интерфейс, 75-76
интерфейсные классы, 909, 911, 913
– функции, 71
интерфейсы, 895
– и реализации, 888
информация о типе объекта на этапе выполнения программы, 494
исключение, 69
– bad_alloc, 180-181, 685
– bad_typeid, 501
– length_error, 709
– out_of_range, 695, 697, 708
– std::bad_alloc, 675
исключения, 219-252, 735, 1077-1114
– bad_alloc, 469
– в деструкторах, 455
– в конструкторах, 452
исключительные ситуации, 68
исходный файл, 253-278, 930
итератор istreambuf_iterator, 666
– ostreambuf_iterator, 666
– ввода, 612
– вывода, 612
– произвольного доступа, 612
итераторы, 82-83, 535, 557, 577, 655-686, 694
– для вставок, 662
– и последовательности, 656-667
– произвольного доступа, 612
– типа const_iterator, 96
– – iterator, 96

К

каркас приложения, 918
категории итераторов, 660
квалификатор, 66
квалифицированные имена, 223
класс allocator, 680
– basic_fstream, 752
– –_ios, 765
– –_istream, 726, 762
– –_istringstream, 755
– –_ostream, 726, 752, 762
– –_streambuf, 760, 762
– –_string, 702
– complex, 107
– date, 1049
– facet, 766, 1008, 1020
– gslice, 796
– ifstream, 752
– istrstream, 756
– locale, 766, 1008
– locale, 764-765, 1011-1019
– ostream, 722
– ostrstream, 756
– reverse_iterator, 664
– string, 89, 362
– valarray, 780
– представления, 363
– complex, 72
классификация символов, 712, 1063
классовая иерархия, 79, 378, 473
классовый шаблон auto_ptr, 447
класс-представление, 352
классы, 114, 284-295, 860-883, 940-941
– для математических вычислений, 775-806
– функциональных объектов, 607-652
ключ, 576
ключевое слово catch, 242
– – const, 141, 290
– – explicit, 353
– – export, 429
– – extern, 255, 266
– – mutable, 294
– – operator, 329
– – protected, 376
– – static, 257
– – struct, 151
– – template, 995
– – this, 292
– – throw, 242
– – try, 243
– – typedef, 130
– – typename, 993
– – void, 62, 200-201
ключевые слова, 125, 151, 926
– – языка C++, 126
код, ориентированный на выражения, 176
кодировка EBCDIC, 116
– Unicode, 117, 690, 1067
кодировки ASCII, 116

комитет X3J16, 46
комментарии, 62, 78, 190-191
комплексные числа, 107
композитные исключения, 440
композиция пространств имен, 235
компоненты, 884-885, 887, 889, 891
компоновка, 254-263, 930
компоновщик, 254
конкатенация, 88, 704
конкретизация, 995
— шаблона, 406
— шаблона функции, 1003
конкретные типы, 73, 74, 301, 303, 895
конкретный тип, 78
— узловой класс, 905
константные выражения, 967
— ссылки, 319
— функции-члены, 290
константный объект, 141, 294
— указатель, 144
константы, 141, 143
конструирование новых объектов локализации, 1015
конструктор, 72-74
конструкторы, 182, 287, 538, 581, 590, 695
— и деструкторы, 376
— и инварианты, 1093
— и преобразования типов, 339
— класса valarray, 780
— по умолчанию, 306
контейнер, 890
— deque, 570
— hash_map, 594
— list, 95
— map, 96, 556
— vector, 93
— типа vector, 533-549
контейнерные классы, 82, 379
контейнерный адаптер, 571
контейнеры, 59, 81, 82, 92-93, 95, 97
— bitset, 743
— STL, 531
— с общим базовым классом, 527
контекст classic(), 1018
— global(), 1018
контроль границ массива, 141
конфликт имен, 231
концепции делегирования, 360
— последовательностей, 83
— сокрытия информации, 66
копирование объектов, 308
— по фактической записи, 365
копирующие алгоритмы, 632
копирующий конструктор, 290, 309, 339, 343, 540, 697
короткие синонимы, 130
косвенное обращение, 134, 360
критерии разрешения перегрузки, 202
куча, 74, 178, 648, 979
кэширование, 293

Л

леводопустимые выражения, 130
лексические соглашения, 927, 929
ленивые вычисления, 295
листовой класс, 905
литералы с плавающей запятой, 119
логическая операция !, 175
— — &&, 175
— — ||, 175
логические операции, 186
логический тип, 115
логическое «И», 173
— «ИЛИ», 173
— false, 185
— постоянство, 292
локализация, 764-765, 1007-1076
локальные объекты, 307
— переменные, 197
— статические объекты, 315

М

макроконстанта NULL, 134
макроопределения, 944
макрос __cplusplus, 263
— arg_two, 216
— assert(), 879
— DBL_MIN_ EXP, 778
— FLT_RADIX, 778
— LDBL_MAX, 778
— va_arg(), 209
— —_start, 209
макросы, 213, 215, 778
— с аргументами, 214
манипуляторы, 741, 743, 745
— без аргументов, 748
— с аргументами, 748
—, определяемые пользователем, 749
—, принимающие аргументы, 746
маски, 797
массив, 64
— mask_array, 797
— slice_array, 789
— указателей, 200
массивы, 64, 114, 135, 137, 973

массивы, valarray, 589
– массивов, 135
– символов, 90
математические функции, 108
машинозависимые аспекты
фундаментальных типов, 121
менеджеры-распределители памяти, 534
метка public, 286
метки, 189
– case-ветвей, 142
механизм RTTI, 473, 493-503
– исключений языка C++, 436
– наследования, 79
многократное использование конкретных типов, 899
многомерные массивы, 135
многострочные комментарии, 78
многоточие, 202, 207
множественное наследование, 392, 473-485
– – и контроль доступа, 492
множество, 97, 588
модификатор const, 182
– const, 134, 198
– explicit, 353
– extern, 262
– friend, 346
– inline, 196
– signed, 118
– static, 197, 346
– volatile, 182
модифицирующие алгоритмы, 632-641
модуль, 65
– stack, 71
модульное программирование, 65, 67, 69
модульность, 59

Н

наследование и шаблоны, 422-423,425
– интерфейса, 871
– классов, 80, 371-400
– реализации, 871
небуферизованный выходной поток, 165
неименованные пространства имен, 257
– пространства имен, 232
неконстантная ссылка, 319
немодифицирующие алгоритмы, 626-631
неперехваченные исключения, 462-463
неявное преобразование типов, 121, 967, 969
нулевой указатель, 134
– элемент, 656

О

область видимости, 127, 128
обобщенное программирование, 59, 61, 81, 83, 401
обобщенные алгоритмы, 82, 521
– аллокаторы, 680
обобщенный код , 83
– срез, 796
обработка исключений, 59, 68, 433-470, 943
– ошибок, 68, 164
обработчик исключений, 68, 243
обратные итераторы, 663
объединение, 50, 321, 977
объект loc, 763
объектно-ориентированное программирование, 59, 60, 77, 79
объекты, 77, 304-321
– locale, 1010
– типа locale, 1007
–-функции, 356,616
объявление, 62, 70
– класса, 285
– переменной внутри условий, 187
– функции, 195
объявления, 123-930, 935-939
– using, 224
– как операторы, 184
– функций, 195
одиночное наследование, 474
оператор break, 91, 187-190, 245
– continue, 190
– do, 188
– for, 188
– goto, 188, 189
– if, 185
– return, 68,187-188,200
– statement, 62
– switch, 64, 91, 142, 158, 162, 185, 186
– throw, 68, 188, 247, 437
– typedef, 117, 692
– while, 188
– цикла for, 65
– – while, 64
операторы, 934
– return, 201
– ветвления, 63
– выбора, 185
– цикла, 188
– языка C++, 183, 185, 187, 189
операции, 62, 895
– sizeof, 120,329

операции static_cast и dynamic_cast, 499
— typeid, 329
— битового сдвига влево и вправо, 158
— и пространства имен, 332
— препроцессора ##, 215
— приведения const_cast, 294
— — типов, 343
— присваивания, 63
— сравнения, 63, 185
— языка C++, 169-181
операционная система UNIX, 42
операция следования, 135
— [], 697
— &, 174-175
— &&, 173
— *, 83
— ::, 128
— |, 175
— ||, 173
— |=, 175
— ++, 83, 175
— +=, 89
— ->, 149
— >>, 63, 164
— delete, 43, 177, 179, 309, 314, 684, 981
— delete [], 179, 314-315
— dynamic_cast, 43, 494-495
— flush(), 758
— new, 43, 50, 74, 177, 309, 684
— — [], 180
— reinterpret_cast, 182
— sizeof, 172
— static_cast, 181
— throw, 43
— try-блок, 43
— typeid, 43, 501
— добавления новых символов в конец строки, 703
— запятая, 173
— инкремента, 83
— инкремента ++, 65
— преобразования const_cast, 182
— — dynamic_cast, 182
— приведения reinterpret_cast, 320
— присваивания, 63, 290, 309, 343
— разрешения области видимости, 128
— разыменования, 83
— — ->, 358
— — указателей на структуры, 149
— сравнения >, 63
— — !=, 63
— — =, 63
— — ==, 63
— — >=, 63
— — >>, 63
— — на равенство, 63
операция точка, 149
определение интерфейсов, 830
— класса, 285
— функции, 196
— шаблона, 404
определения, 124
— функций, 196
организация стандартной библиотеки, 520
отложенное выделение ресурсов, 1097
отношение «дружбы», 989
— включения, 860, 867
— использования, 860, 874
— типа "иметь", 869
отношения внутри класса, 860, 877
— наследования, 860
отображение, 576
— исключений, 460
отображенное значение, 576
отрицатели, 625
очередь, 97, 572
— с приоритетом, 97, 574
очистка памяти, 73
ошибки выделения памяти, 956
— на этапе выполнения, 244

П

парадигма программирования, 59, 60, 65
параметр типа, 81
— шаблона, 81
параметризация и наследование, 424
параметрический полиморфизм, 424
параметры шаблонов, 406
— — по умолчанию, 415
паттерн специализации, 418
паттерны, 833
перегруженная операция присваивания, 540
перегруженные операции, 303
перегрузка, 942
— имен функций, 201, 203, 205
— операций, 303, 327-370
— функциональных шаблонов, 411
передача аргументов, 197, 199
перекрестное приведение, 495
перестановки, 650
перехват исключений, 441, 443

перехват любых исключений, 442
перечисления, 122
побитовая логическая операция «И», 158
– – – «ИЛИ», 158
– – – «ИСКЛЮЧАЮЩЕЕ ИЛИ», 158
побитовое копирование, 308
побитовые логические операции, 174
– операции, 591
повторная генерация исключений, 441
повторное использование кода, 838
– – проектов, 833
повышающее приведение, 495
поддержка парадигмы программирования, 60
подкласс, 373
подстроки, 707
подтипизация, 871
поисковые алгоритмы, 631
поисковый алгоритм find_end(), 631
– алгоритм search(), 631
– – search_n(), 631
поле типа, 78
полиморфизм, 383
– времени выполнения, 424
– – компиляции, 424
полиморфные процедуры, 211
полиморфный тип, 75, 383
полное выражение, 318
пользовательские предикаты, 618
– специализации, 417
пользовательский аллокатор, 678
– тип, 42
понижающее приведение, 494
порядок вычисления аргументов вызова функций, 173
– – подвыражений, 172
– записи обработчиков, 443
– специализаций, 420
последовательность элементов, 82, 98
последовательные контейнеры, 555, 565-569
поток cerr, 722
– cin, 162, 726
– clog, 722
– cout, 87, 722
– istream, 729, 757
– stderr, 722
– stdin, 726
– stdout, 722
– ввода cin, 726
– – wcin, 726
потоки, 717-772, 1109
– типа ostream, 174
потоковые итераторы, 664
почленное копирование, 308
правила ассоциативности, 173
правило «неявного int», 126
– одного определения, 260
предварительное объявление, 151
предикатные функции, 115
предикаты, 103, 115, 617, 1107
предкомпиляция заголовочных файлов, 258
представления, 914
предусловия и постусловия, 882
преобразования указателей и ссылок, 969
– – на члены классов, 970
– чисел с плавающей запятой, 969
– шаблонов, 426
препроцессор, 944
препроцессорные директивы #include, 236
префикс L, 138
– std::, 166
– templateclass C, 403
приведение типов, 181
– – в функциональном стиле, 183
прикладные среды разработки, 895, 918-919
принцип подстановки Liskov, 871
– раздельной компиляции, 67
– сокрытия данных, 65
приоритет, 171
– операций, 173
приращения, 803
пробельные символ, 119ы, 164
проверка диапазона индексов, 94
программируемые отношения, 860, 875
продвижения, 968
– типов вверх, 202
проектирование, 849-894
– иерархий классов, 386-395
производные исключения, 438
– классы, 372-383, 941
пространства имен, 66, 219-252, 955, 983
– имен std, 87, 98, 166, 238, 609, 691, 746-747, 798
прототипирование, 835
процедурное программирование, 59, 61, 63
прямой итератор, 612
псевдонимы, 234
пунктуация чисел, 1033
пустая строка, 137

Р

раздельная компиляция, 59, 67, 68, 253
размещающее new, 320
разновидности классов, 895
разрешение перегрузки функций, 203
разыменование, 134
раскрутка стека, 436, 446
распределители памяти, 674-685
расширенная информация о типе, 502
расширенные символьные наборы, 965
реализация интерфейса, 76
– модуля, 70
редуцированная схема вычислений, 173
рекурсивный спуск, 156
ромбовидное наследование, 484

С

свободная память, 178, 509, 511
связывание имен, 996
связывающие адаптеры, 621, 625
семантика владения, 448
– возврата значения из функции, 200
– деструктивного копирования, 448
– передачи аргументов, 195, 198
символ, 86, 117
– «перевода строки», 86
– n, 63
символы, 690
символьная константа, 88
символьные литералы, 117
– наборы, 963, 965
– типы, 116-117
синтаксический анализатор, 156
скалярное произведение, 803
словарь, 97, 576
служебный символ end-of-file, 166
событие copyfmt_event, 766
– erase_event, 766
– imbue_event, 766
совместимость C и C++, 948-949, 951
– с языком C, 42
соглашение компоновки, 262
– о компоновке, 263
сокрытие имен, 128
сопровождение и поддержка программ, 837
сортировка последовательностей, 643, 645, 647
составные (комбинированные) операции присваивания, 158
состояние, 729
– смещения, 1067
специализации, 406
–, определяемыми пользователем, 417
специализация numeric_limitschar, 777
– vectorbool, 550
– шаблонов функций, 420
спецификатор доступа private, 295
спецификаторы, 125
спецификации языка C++, 46
спецификация исключений, 457, 459, 461
списки, 45
список, 95, 565
– инициализации членов, 310
средство вывода, 87
срез, 786
ссылки, 144-145, 147
ссылочные типы, 114
стандарт ANSI C, 49
– C++, 947
стандартная библиотека, 85-108, 517-523, 955
– – языка C++, 82
– библиотечная функция atexit(), 277
– – – isalpha(), 162
– – – isspace(), 164
– функция set_new_handler(), 180
стандартные алгоритмы, 607
– арифметические преобразования, 171
– исключения, 467
– контейнеры, 97, 555-604
– макросы, 208
– манипуляторы ввода/вывода, 747
– преобразования типов, 202
– фасеты, 1026-1073
стандартный алгоритм find(), 99
– аллокатор, 674
– библиотечный класс string, 689
– заголовочный файл, 87
– класс фасетов collate_bynamechar, 1020
– поток ввода, 90
– – вывода, 63,88
– – – std::cout, 86
– – ошибок cerr, 95
– строковый шаблон basic_string, 692-709
– тип string, 305
– шаблон allocator, 674
статическая функция-член, 289
статические переменные, 197
– члены, 288-289, 990
– –-константы, 312
статический объект, 129
стек, 65, 97, 571
стековая память, 979
стековые операции, 541

стиль программирования, 40
строгое слабое упорядочение, 562
строки, 88-89, 689-714, 1108
– типа string, 589
– языка C, 90
строковые потоки, 755
строковый литерал, 86, 136
структуры, 149, 151
– данных, 114
– и классы, 295
суперкласс, 373
схема, форма, 418
счетчик ссылок, 362
сырая память, 181, 501, 676

Т

таблица виртуальных функций, 77
тело определения класса, 285
– функции, 62
терминальный нуль, 90
тестирование, 836
тип, 62, 113
– allocator_type, 534
– bool, 115
– char, 62,115-116
– –*, 207
– double, 72, 91, 115
– ifstream, 103
– indirect_array, 798
– int, 62, 115
– istream, 90
– istream_iterator, 103
– long, 171
– mask, 1063
– –_array, 797
– ofstream, 103
– ostream, 90
– –_iterator, 103
– out_of_range, 94
– pair, 104
– rebind, 677
– size_t, 510
– –_type, 697
– streamsize, 721
– string, 88, 90, 92
– time_t, 1048
– vector, 50, 94
– void, 121
– –*, 148
– wchar_t, 117, 692
– аллокаторов, 534
– итератора, 96
– стандартной библиотеки, 42
–, определяемый пользователем, 284
типы, 113
– valarray и complex, 1110
– итераторов, 101
– с плавающей запятой, 114, 119
– стандартной библиотеки, 87
– целых литералов, 967
–, определяемые пользователем, 71, 114
–-значения, 303
точка входа в программу, 86
– конкретизации шаблона, 409
– объявления, 128
точное совпадение типов, 202

У

удобства функциональной декомпозиции, 43
узловой класс иерархии, 903
узловые классы, 903,905,907
узлы, 895
указатели, 64, 133
– на массивы, 133, 138-139
– на функции, 133, 148, 209, 211
– – члены классов, 148, 473, 505, 507, 989
указатель, 65
– _unexpected_handler, 462
– this, 346
– на член класса, 507
указательные типы, 114
унарная операция, 330
– – &, 65
унарный предикат, 612, 617
ундаментальные типы, 114
управление памятью, 979, 981
– режимом доступа, 285
управляющая строка, 207
уровни приоритетов операций, 173
условная компиляция, 215
условные выражения, 186
– операторы, 185
утверждения, 879
утечка памяти, 310

Ф

фабрика, 396
файл cstdarg, 208
– cstdio, 238
– exception, 462
– functional, 617, 619, 622
– ios, 721

файл iostream, 86, 721-722, 726
— limits , 121
— stack, 571
— stdexcept, 94
— stdio.h, 238
— typeinfo, 501
— vector, 533
фасет _byname(), 1021
— codecvt, 1068
— collate_byname, 1032
— money_get, 1040, 1041, 1044
— —_put, 1040-1041, 1043
— moneypunct, 1040
— numpunct, 1033
— time_get, 1052
— —_put, 1049
фасеты, 1007, 1020-1025
фигурные скобки, 62
флаг uppercase, 743
флаги состояния потока, 175
формальная грамматика, 156
форматирующий флаг boolalpha, 724
функторы, 356, 616
функции, 195-218
— exit(), 277
— insert(), 96
— get(), 709
— operator new(), 674
— printf(), 207
— seekp(), 757
— strcmp(), 692
— доступа, 531
— обратного вызова для потоков, 766
— поддержки, 302, 341
—-операции, 329, 331, 333
функции-члены, 284, 285, 300
— operator*(), 328
— —+(), 328
функциональные объекты, 607, 616
функциональный шаблон use_facet, 1021
функция abort(), 277
— apply(), 783
— at(), 94, 695
— atexit(), 277
— begin(), 96,99
— bsearch(), 650
— c_str(), 90, 700
— clear(), 735
— clock(), 1047
— close(), 753
— complete_name(), 704
— data(), 700
— egptr(), 761
— end(), 96,99
— eof(), 734
— eq(), 600
— error(), 208
— fill(), 744
— flags(), 740
— frac_digits() фасета moneypunct, 1040
— get_allocator(), 550
— —_time(), 1053
— getline(), 709, 731, 732
— hash(), 600
— ignore(), 733
— imbue(), 759,765
— imbue(), 765
— in_avail(), 759
— itoa(), 208
— iword(), 766
— main(), 86, 165, 275
— narrow(), 760
— operator delete [] (), 511
— — delete(), 321
— — new [] (), 511
— — —(), 180-181, 320
— —()(), 600
— —==(), 303
— precision(), 743
— put(), 722
— putback(), 758
— pword(), 766
— qsort(), 650
— readsome(), 759
— register_callback(), 766
— replace(), 706
— reserve(), 548
— setf(), 740, 744
— setg(), 761
— setp(), 761
— showmanyc(), 763
— strchr(), 711
— strtod(), 712
— substr(), 707
— swap(), 709
— swap(), 1105
— tie(), 737
— tolower(), 1065
— toupper(), 1065
— uflow(), 763
— underflow(), 763
— unget(), 758
— widen(), 760
— width(), 729
— width(), 743
— write(), 722

функция xalloc(), 766
— выделения памяти operator new(), 320
— обратного вызова, 388
—-операции operator()(), 356
—-операция operator [], 355
функция-член assign(), 540
— operator delete(), 509
— — new(), 509

Х–Ц–Ч

хэширование, 600
хэш-таблица, 598
целые литералы, 118
— типы, 118
цикл do, 189
— while, 69, 189
циклы, 63-64
часовой, 738
частичная специализация, 418
числовые пределы, 776-777

Ш–Э–Я

шаблон, 81
— Assert(), 880
— auto_ptr, 520
— basic_ostream, 720-721
— char_traits, 690
— complex, 798
— iterator_traits, 634,658-659
— mem_fun(), 106
— numeric_limits, 776
шаблонные члены шаблонов, 424
шаблоны, 45, 50, 401-432, 886, 942, 956, 990-1003
— и пространства имен, 1001
— языка C++, 61
шаг, 787
эквивалентность типов, 152
элемент-часовой, 83
явная конкретизация, 1004
явное управление памятью, 1087
явный вызов деструктора, 321
язык C, 42, 49

Научно-техническое издание

Бьерн Страуструп

Язык программирования C++. Специальное издание

Подписано в печать 13.09.2010. Формат $70\times100/_{16}$. Усл. печ. л. 92,3.
Гарнитура Таймс. Бумага газетная. Печать офсетная.
Тираж 2000 экз. Заказ 10828

Издательский дом Бином,
127018, Москова, ул. 1-я Ямская, 8

При участии ООО «Столица–Принт»
e-mail: st.print@bk.ru

Отпечатано с готовых файлов заказчика
в АО «Первая Образцовая типография»,
филиал «УЛЬЯНОВСКИЙ ДОМ ПЕЧАТИ»
432980, г. Ульяновск, ул. Гончарова, 14

БОЛЕЕ МИЛЛИОНА ПРОГРАММИСТОВ ВО ВСЕМ МИРЕ ПРИОБРЕЛИ ПРЕДЫДУЩИЕ ИЗДАНИЯ ЭТОЙ КНИГИ

Перед вами – специальное издание самой читаемой и содержащей наиболее достоверные сведения книги по C++. В него были добавлены два новых приложения: "***Локализация***" и "***Безопасность исключений и стандартная библиотека***" (оба доступны на сервере http://www.research.att.com/~bs). В результате монография содержит полное описание языка C++, его стандартной библиотеки (STL) и ключевых методов разработки программ. Основанная на стандарте ANSI/ISO, книга является источником самого последнего и полного описания всех возможностей языка C++, включая компоненты стандартной библиотеки, в том числе:

- абстрактные классы в качестве интерфейсов;
- иерархию классов при объектно-ориентированном программировании;
- шаблоны как основу безопасного относительно типов обобщенного программирования;
- обработку исключений, возникающих в результате типичных ошибок;
- использование пространств имен (namespaces) для достижения модульности больших проектов;
- определение типа на этапе исполнения (RTTI) для слабо связанных систем;
- подмножество C языка C++ для совместимости с C и работы на системном уровне;
- стандартные контейнеры и алгоритмы;
- стандартные строки, потоки ввода/вывода и числовые данные;
- совместимость с C, локализацию (интернационализацию) и безопасность при обработке исключениий.

В данном издании Бьерн Страуструп излагает C++ в форме, еще более доступной для начинающих. В то же время в него включены такие сведения и методики, которые могут оказаться полезными даже для экспертов по C++.

Web-поддержка книги осуществляется по адресу: http://www.research.att.com/~bs

Бьерн Страуструп (***Bjarne Stroustrup***) непосредственно разработал и первым осуществил реализацию языка программирования C++. Он – автор книг: "***Язык программирования C++***" (***The C++ Programming Language***, первое издание – 1985, второе – 1991, третье – 1997), "***Справочное руководство по языку C++ с комментариями***" (***The Annotated C++ Reference Manual***) и "***Проектирование и эволюция C++***" (***The Design and Evolution of C++***). Закончил университет в Аархусе (Дания) и Кэмбриджский университет (Англия). В настоящее время д-р Страуструп возглавляет Отдел исследований в области крупномасштабного программирования компании AT&T Labs и является членом совета AT&T, AT&T Bell Laboratories и ACM. Область его научных интересов – распределенные системы, операционные системы, моделирование, проектирование и программирование. Он является редактором серии **C++ In-Depth**, выпускаемой издательством Addison-Wesley.

ISBN 978-5-9518-0425-9
9 785951 804259