Антон Полухин

Разработка приложений на C++ с использованием Boost

Рецепты, упрощающие разработку вашего приложения

Boost C++ Application Development Cookbook

Recipes to simplify
your application development

Antony Polukhin

Разработка приложений на C++ с использованием Boost

Рецепты, упрощающие разработку вашего приложения

Антон Полухин

Москва, 2020

УДК 004.4
ББК 32.973.202
П49

Антон Полухин
П49 Разработка приложений на С++ с использованием Boost. Рецепты, упрощающие разработку вашего приложения / пер. с анг. Д. А. Беликова. – М.: ДМК Пресс, 2020. – 346 с.: ил.

ISBN 978-5-97060-868-5

Это руководство знакомит читателя с библиотеками Boost, которые помогают разрабатывать качественные, быстрые и портативные приложения. Удобная структура книги, включающая ряд стандартных разделов, упрощает изучение материала. От простых тем (повседневное использование библиотек, управление ресурсами) автор последовательно переходит к сложным (метапрограммирование, многопоточность, межпроцессное взаимодействие, асинхронное взаимодействие, работа с большими библиотеками Boost).

Издание предназначено для разработчиков, желающих улучшить свои знания в области Boost и упростить процессы разработки приложений. Для освоения изложенных в книге приемов необходимы знакомство с С++ и базовые знания стандартной библиотеки. Также понадобятся современный компилятор С++, библиотеки Boost (подойдет любая версия, но рекомендуется 1.65 или более новая), среда разработки QtCreator, утилита qmake. Есть возможность модифицировать и запускать примеры онлайн: http://apolukhin.github.io/Boost-Cookbook/.

УДК 004.4
ББК 32.973.202

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Все права защищены. Любая часть этой книги не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами без письменного разрешения владельцев авторских прав.

ISBN (анг.) 978-1-78728-224-7
ISBN (рус.) 978-5-97060-868-5
© 2017 Packt Publishing
© Оформление, издание, перевод, ДМК Пресс, 2020

Оглавление

Предисловие от издательства 20
Об авторе 21
О рецензентах 22
Вступительное слово автора 23
Вступительное слово от сообщества C++ 24
Предисловие 25

Глава 1. Приступаем к написанию приложения 29
Вступление 29
Получение параметров конфигурации 30
Подготовка 30
Как это делается 30
Как это работает 31
Дополнительно… 31
См. также 33
Сохранение любого значения в контейнере или переменной 33
Подготовка 34
Как это делается 34
Как это работает 34
Дополнительно… 35
См. также 36
Хранение одного из нескольких выбранных типов в контейнере или переменной 36
Подготовка… 36
Как это делается 36
Как это работает 37
Дополнительно… 37
См. также 38
Использование более безопасного способа работы с контейнером, в котором хранится один из нескольких выбранных типов 38
Подготовка 40
Как это делается 40
Как это работает 41
Дополнительно… 41
См. также 42

Возврат значения или флага «значения нет» 42
Подготовка 42
Как это делается........ 42
Как это работает........ 43
Дополнительно... 44
См. также 44
Возвращение массива из функции........ 44
Подготовка 44
Как это делается........ 44
Как это работает........ 45
Дополнительно... 45
См. также 46
Объединение нескольких значений в одно 46
Подготовка 46
Как это делается........ 46
Как это работает........ 48
Дополнительно... 48
См. также 48
Привязка и переупорядочение параметров функции 49
Подготовка 49
Как это делается........ 49
Как это работает........ 50
Дополнительно... 51
См. также 52
Получение удобочитаемого имени типа 52
Подготовка 52
Как это делается........ 52
Как это работает........ 53
Дополнительно... 53
См. также 54
Использование эмуляции перемещения C++11 54
Подготовка 54
Как это делается........ 54
Как это работает........ 56
Дополнительно... 56
См. также 56
Создание некопируемого класса 57
Подготовка 57
Как это делается........ 57
Как это работает........ 58
См. также 58
Создание некопируемого, но перемещаемого класса 58
Подготовка 59
Как это делается........ 59
Как это работает........ 61
Дополнительно... 61
См. также 62

Использование алгоритмов C++14 и C++11 62
Подготовка 62
Как это делается 63
Как это работает 63
Дополнительно... 63
См. также 64

Глава 2. Управление ресурсами 65
Вступление 65
Управление указателями на классы, которые не покидают область видимости 65
Подготовка 66
Как это делается 66
Как это работает 67
Дополнительно... 68
См. также 69
Подсчет указателей на классы 69
Подготовка 69
Как это делается 70
Как это работает 70
Дополнительно... 71
См. также 72
Управление указателями на массивы, которые не покидают область видимости 72
Подготовка 72
Как это делается 73
Как это работает 73
Дополнительно... 73
См. также 74
Подсчет указателей на массивы 74
Подготовка 75
Как это делается 75
Как это работает 77
Дополнительно... 77
См. также 77
Хранение любых функциональных объектов в переменной 78
Подготовка 78
Как это делается 78
Как это работает 79
Дополнительно... 79
См. также 80
Передача указателя на функцию 80
Подготовка 80
Как это делается 80
Как это работает 80
Дополнительно... 80
См. также 81

Хранение любых лямбда-функций C++11 в переменной 81
Подготовка 81
Как это делается....................81
Дополнительно... 81
См.также 82
Контейнеры указателей 82
Подготовка 84
Как это делается....................84
Как это работает....................84
Дополнительно... 84
См. также 85
Делайте это при выходе из области видимости! 86
Подготовка 86
Как это делается....................86
Как это работает....................87
Дополнительно... 87
См. также 88
Инициализация базового класса членом класса-наследника 88
Подготовка 89
Как это делается....................89
Как это работает....................89
Дополнительно... 90
См. также 90

Глава 3. Преобразование и приведение 91

Вступление 91
Преобразование строк в числа 91
Подготовка 92
Как это делается....................92
Как это работает....................92
Дополнительно... 93
См. также 94
Преобразование чисел в строки 94
Подготовка 94
Как это делается....................94
Как это работает....................95
Дополнительно... 95
См. также 96
Преобразование чисел в числа 96
Подготовка 97
Как это делается....................97
Как это работает....................98
Дополнительно... 98
См. также 99
Преобразование пользовательских типов в строки и из строк....................99
Как это делается....................99
Дополнительно... 101
См. также 101

Приведение умных указателей ... 102
Подготовка ... 102
Как это делается ... 102
Как это работает ... 103
Дополнительно… ... 103
См. также ... 103
Приведение полиморфных объектов ... 103
Подготовка ... 104
Как это делается ... 104
Как это работает ... 104
Дополнительно… ... 104
См. также ... 105
Синтаксический анализ (parsing) простого ввода ... 105
Подготовка ... 106
Как это делается ... 106
Как это работает ... 107
Дополнительно… ... 108
См. также ... 110
Синтаксический анализ (parsing) сложного ввода ... 110
Подготовка ... 110
Как это делается ... 110
Как это работает… ... 113
Дополнительно… ... 114
См. также ... 114

Глава 4. Уловки времени компиляции ... 115

Вступление ... 115
Проверка размеров во время компиляции ... 115
Подготовка ... 116
Как это делается ... 116
Как это работает ... 116
Дополнительно… ... 118
См. также ... 119
Активация использования шаблона функции для интегральных типов ... 120
Подготовка ... 120
Как это делается ... 120
Как это работает ... 121
Дополнительно… ... 122
См. также ... 123
Отключение использования шаблона функции для действительных типов ... 123
Подготовка ... 124
Как это делается ... 124
Как это работает ... 124
Дополнительно… ... 125
См. также ... 125
Создание типа из числа ... 125
Подготовка ... 126

Как это делается 126
Как это работает 127
Дополнительно… 127
См. также 128
Реализация свойства типов (type trait) 128
Подготовка 128
Как это делается 128
Как это работает 128
Дополнительно… 129
См. также 129
Выбор оптимального оператора для параметра шаблона 129
Подготовка 130
Как это делается 130
Как это работает 131
Дополнительно… 131
См. также 133
Получение типа выражения в С++03 133
Подготовка 133
Как это делается 133
Как это работает 134
Дополнительно… 134
См. также 135

Глава 5. Многопоточность 137

Вступление 137
Создание потока выполнения 137
Подготовка 138
Как это делается 138
Как это работает 138
Дополнительно… 139
См. также 141
Синхронизация доступа к общему ресурсу 141
Подготовка 142
Как это делается 142
Как это работает 143
Дополнительно… 144
См. также 145
Быстрый доступ к общему ресурсу с использованием атомарных операций 145
Подготовка 145
Как это делается 146
Как это работает 146
Дополнительно… 147
См. также 148
Создание класса work_queue 148
Подготовка 148
Как это делается 148

Как это работает....150
Дополнительно... 151
См. также 152
Блокировка «Несколько читателей – один писатель»....152
Подготовка 153
Как это делается....154
Как это работает....154
Дополнительно... 155
См. также 155
Создание переменных, уникальных для каждого потока 155
Подготовка 156
Как это делается....156
Как это работает....157
Дополнительно... 157
См. также 157
Прерывание потока 157
Подготовка 158
Как это делается....158
Как это работает....158
Дополнительно... 159
См. также 159
Манипулирование группой потоков 159
Подготовка 160
Как это делается....160
Как это работает....160
Дополнительно... 160
См. также 161
Безопасная инициализация общей переменной 161
Подготовка 162
Как это делается....162
Как это работает....163
Дополнительно... 163
См. также 164
Захват нескольких мьютексов 164
Подготовка 165
Как это делается....165
Как это работает....165
Дополнительно... 166
См. также 166

Глава 6. Манипулирование задачами 167

Вступление 167
Прежде чем вы начнете....167
Регистрация задачи для обработки произвольного типа данных 168
Подготовка 168
Как это делается....168

Как это работает..170
Дополнительно...171
См. также...171
Создание таймеров и обработка событий таймера в качестве задач..........172
Подготовка..172
Как это делается..172
Как это работает..174
Дополнительно...175
См. также...176
Передача данных по сети в качестве задачи...............176
Подготовка..176
Как это делается..176
Как это работает..180
Дополнительно...183
См. также...184
Прием входящих соединений...............................184
Подготовка..184
Как это делается..184
Как это работает..186
Дополнительно...187
См. также...188
Параллельное выполнение различных задач.................188
Приступаем..188
Как это делается..188
Как это работает..189
Дополнительно...190
См. также...190
Конвейерная обработка задач.............................190
Подготовка..191
Как это делается..191
Как это работает..193
Дополнительно...194
См. также...194
Создание неблокирующего барьера.........................194
Подготовка..196
Как это делается..196
Как это работает..197
Дополнительно...197
См. также...198
Хранение исключения и создание задачи из него...........198
Подготовка..198
Как это делается..198
Как это работает..200
Дополнительно...200
См. также...201
Получение и обработка системных сигналов в качестве задач..............201
Подготовка..202

Как это делается..202
Как это работает..203
Дополнительно… 204
См. также 204

Глава 7. Манипулирование строками 205
Вступление 205
Смена регистра символов и сравнение без учета регистра........................205
Подготовка 205
Как это делается..206
Как это работает..207
Дополнительно… 207
См. также 207
Сопоставление строк с использованием регулярных выражений....................207
Приступим 208
Как это делается..208
Как это работает..210
Дополнительно… 210
См. также 211
Поиск и замена строк с использованием регулярных выражений....................211
Подготовка 212
Как это делается..212
Как это работает..213
Дополнительно… 213
См. также 213
Форматирование строк с использованием безопасных
printf-подобных функций 214
Подготовка 214
Как это делается..214
Как это работает..215
Дополнительно… 215
См. также 216
Замена и стирание строк..216
Подготовка 216
Как это делается..216
Как это работает..217
Дополнительно… 217
См. также 218
Представление строки двумя итераторами..218
Подготовка 218
Как это делается..218
Как это работает..219
Дополнительно… 220
См. также 220
Использование типа «ссылка на строку» 221
Подготовка 221
Как это делается..221

Как это работает... 222
Дополнительно... 224
См. также... 224

Глава 8. Метапрограммирование... 225

Вступление... 225
Использование типа «вектор типов»... 225
Подготовка... 226
Как это делается... 226
Как это работает... 228
Дополнительно... 229
См. также... 229
Манипулирование вектором типов... 230
Подготовка... 230
Как это делается... 230
Как это работает... 232
Дополнительно... 233
См. также... 234
Получение результирующего типа функции во время компиляции... 234
Подготовка... 235
Как это делается... 235
Как это работает... 236
Дополнительно... 236
См. также... 236
Создание метафункции высшего порядка... 236
Подготовка... 237
Как это делается... 237
Как это работает... 238
Дополнительно... 238
См. также... 239
Ленивое вычисление метафункции... 239
Подготовка... 239
Как это делается... 239
Как это работает... 240
Дополнительно... 242
См. также... 242
Преобразование всех элементов кортежа в строки... 242
Подготовка... 242
Как это делается... 242
Как это работает... 244
Дополнительно... 244
См. также... 246
Расщепление кортежей... 246
Подготовка... 246
Как это делается... 246
Как это работает... 247

Дополнительно… 248
См. также 249
Манипулирование гетерогенными контейнерами в C++14 249
Подготовка 249
Как это делается 249
Как это работает 250
Дополнительно… 251
См. также 252

Глава 9. Контейнеры 253
Вступление 253
Хранение нескольких элементов в контейнере 253
Подготовка 254
Как это делается 254
Как это работает 255
Дополнительно… 255
См. также 256
Хранение не более ***N*** элементов в контейнере 256
Подготовка 256
Как это делается 256
Как это работает 256
Дополнительно… 257
См. также 258
Сверхбыстрое сравнение строк 258
Подготовка 259
Как это делается 259
Как это работает 260
Дополнительно… 261
См. также 262
Использование неупорядоченных ассоциативных контейнеров 262
Подготовка 262
Как это делается 262
Как это работает 263
Дополнительно… 265
См. также 265
Создание ассоциативного контейнера с индексированием и по значениям 266
Подготовка 266
Как это делается 266
Как это работает 268
Дополнительно… 269
См. также 269
Использование многоиндексных контейнеров 269
Подготовка 270
Как это делается 270
Как это работает 272
Дополнительно… 274
См. также 274

Получение преимуществ от односвязного списка и пула памяти 274
Подготовка 274
Как это делается... 274
Как это работает... 276
Дополнительно... 277
См. также 277
Использование плоских ассоциативных контейнеров 278
Подготовка 278
Как это делается... 278
Как это работает... 279
Дополнительно... 280
См. также 281

Глава 10. Сбор информации о платформе и компиляторе 283
Вступление 283
Обнаружение ОС и компилятора 284
Подготовка 284
Как это делается... 284
Как это работает... 284
Дополнительно... 285
См. также 285
Обнаружение поддержки 128-битных целых чисел 285
Подготовка 285
Как это делается... 285
Как это работает... 286
Дополнительно... 286
См. также 287
Обнаружение и обход отключенной динамической идентификации типа данных 287
Подготовка 287
Как это делается... 287
Как это работает... 288
Дополнительно... 289
См. также 289
Написание метафункций с использованием более простых методов 290
Подготовка 290
Как это делается... 290
Как это работает... 291
Дополнительно... 291
См. также 291
Уменьшение размера кода и повышение производительности пользовательских типов в C++11 292
Подготовка 292
Как это делается... 292
Как это работает... 293
Дополнительно... 293
См. также 293

Переносимый способ экспорта и импорта функций и классов 294
Подготовка 294
Как это делается..................... 294
Как это работает..................... 295
Дополнительно… 296
См. также 296
Обнаружение версии Boost и получение новейших функций 297
Подготовка 297
Как это делается..................... 297
Как это работает..................... 298
Дополнительно… 298
См. также 298

Глава 11. Работа с системой..................... 299
Вступление 299
Перечисление файлов в каталоге 299
Подготовка 300
Как это делается..................... 300
Как это работает..................... 301
Дополнительно… 301
См. также 302
Стирание и создание файлов и каталогов 302
Подготовка 302
Как это делается..................... 302
Как это работает..................... 303
Дополнительно… 304
См. также 304
Написание и использование плагинов 304
Подготовка 304
Как это делается..................... 304
Как это работает..................... 305
Дополнительно… 306
См. также 306
Получение backtrace – текущей последовательности вызовов 306
Приступим..................... 307
Как это делается..................... 307
Как это работает..................... 308
Дополнительно… 308
См. также 309
Быстрая передача данных из одного процесса в другой 309
Подготовка 309
Как это делается..................... 309
Как это работает..................... 310
Дополнительно… 311
См. также 311
Синхронизация межпроцессного взаимодействия 312
Подготовка 312

Как это делается...312
Как это работает...314
Дополнительно...314
См. также...315
Использование указателей в общей памяти...315
Подготовка...315
Как это делается...315
Как это работает...316
Дополнительно...317
См. также...317
Самый быстрый способ чтения файлов...317
Подготовка...317
Как это делается...317
Как это работает...318
Дополнительно...319
См. также...319
Сопрограммы – сохранение состояния и откладывание выполнения...319
Подготовка...319
Как это делается...320
Как это работает...321
Дополнительно...322
См. также...323

Глава 12. Касаясь верхушки айсберга ...325

Вступление...325
Работа с графами...326
Подготовка...326
Как это делается...326
Как это работает...327
Дополнительно...329
См. также...329
Визуализация графов...329
Подготовка...330
Как это делается...330
Как это работает...331
Дополнительно...331
См. также...332
Использование генератора истинно случайных чисел...332
Приступаем...332
Как это делается...332
Как это работает...333
Дополнительно...333
См. также...334
Использование переносных математических функций...334
Подготовка...334
Как это делается...334
Как это работает...334

Дополнительно… 335
См. также 335
Написание тестовых случаев 335
Подготовка 336
Как это делается 336
Как это работает 336
Дополнительно… 337
См. также 337
Объединение нескольких тестовых случаев в одном тестовом модуле 338
Подготовка 338
Как это делается 338
Как это работает 339
Дополнительно… 339
См. также 339
Манипулирование изображениями 339
Подготовка 340
Как это делается 340
Как это работает 342
Дополнительно… 343
См. также 343

Предметный указатель 344

Предисловие от издательства

Отзывы и пожелания

Мы всегда рады отзывам наших читателей. Расскажите нам, что вы думаете об этой книге – что понравилось или, может быть, не понравилось. Отзывы важны для нас, чтобы выпускать книги, которые будут для вас максимально полезны.

Вы можете написать отзыв прямо на нашем сайте **www.dmkpress.com**, зайдя на страницу книги, и оставить комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу **dmkpress@gmail.com**, при этом напишите название книги в теме письма.

Если есть тема, в которой вы квалифицированы, и вы заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу **http://dmkpress.com/authors/publish_book/** или напишите в издательство по адресу **dmkpress@gmail.com**.

Список опечаток

Хотя мы приняли все возможные меры для того, чтобы удостовериться в качестве наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг – возможно, ошибку в тексте или в коде, – мы будем очень благодарны, если вы сообщите нам о ней. Сделав это, вы избавите других читателей от расстройств и поможете нам улучшить последующие версии этой книги.

Если вы найдете какие-либо ошибки в коде, пожалуйста, сообщите о них главному редактору по адресу **dmkpress@gmail.com**, и мы исправим это в следующих тиражах.

Нарушение авторских прав

Пиратство в интернете по-прежнему остается насущной проблемой. Издательство «ДМК Пресс» очень серьезно относится к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в интернете с незаконно выполненной копией любой нашей книги, пожалуйста, сообщите нам адрес копии или веб-сайта, чтобы мы могли применить санкции.

Пожалуйста, свяжитесь с нами по адресу электронной почты **dmkpress@gmail.com** со ссылкой на подозрительные материалы.

Мы высоко ценим любую помощь по защите наших авторов, помогающую нам предоставлять вам качественные материалы.

Об авторе

Если вам интересно, кто такой Антон Полухин и можно ли доверять ему в вопросах обучения C++ и библиотекам Boost, то вот несколько фактов:

- Антон Полухин в настоящее время представляет Россию в международном комитете по стандартизации C++;
- он является автором нескольких библиотек Boost и поддерживает ряд старых библиотек Boost;
- он перфекционист: все исходные коды из книги проходят автоматическое тестирование на нескольких платформах с использованием различных стандартов C++.

Но давайте начнем с самого начала.

Антон Полухин родился в России. В детстве он мог говорить на русском и венгерском языках и изучал английский в школе. Со школьных лет участвовал в различных соревнованиях по математике, физике и химии и побеждал в них.

Дважды был принят в университет: один раз за участие в городской олимпиаде по математике и второй раз за то, что получил высокий балл по вступительным олимпиадам в вуз. В его университетской жизни был год, когда он вообще не участвовал в экзаменах: он получил «зачет автоматом» во всех дисциплинах, написав программы повышенной сложности по каждому предмету. Свою будущую жену он встретил в университете, который закончил с отличием.

Более трех лет работал в VoIP-компании, разрабатывая бизнес-логику для коммерческой альтернативы Asterisc. В то время он начал вносить свой вклад в Boost и стал сопровождающим библиотеки `Boost.LexicalCast`. Он также начал делать переводы на русский язык для Ubuntu Linux в то время.

Сегодня работает в компании Yandex.Taxi, помогает русскоговорящим людям с предложениями по стандартизации C++, продолжает вносить вклад в opensource-проекты и язык C++ в целом.

Его код можно найти в библиотеках Boost, таких как Any, Conversion, DLL, LexicalCast, Stacktrace, TypeTraits, Variant и др.

Счастлив в браке уже более семи лет.

Я хотел бы поблагодарить свою семью, особенно мою жену Ирину Полухину, за то, что она рисовала эскизы рисунков и диаграмм для этой книги.

Огромное спасибо Полу Энтони Бристоу за обзор первого издания данной книги и за то, что он прошел через безумное количество запятых, которые я использовал в первых черновиках.

Отдельное спасибо Глену Джозефу Фернандесу за то, что он предоставил много полезной информации и комментариев по второму изданию.

Что касается русского издания книги – неоценимую помощь оказал Кирилл Марков. За что ему отдельное спасибо!

Я также хотел бы поблагодарить всех членов сообщества Boost за написание этих замечательных библиотек и за то, что они открыли для меня удивительный мир C++.

О рецензентах

Глен Джозеф Фернандес работал инженером-программистом в компаниях Intel и Microsoft. Он является автором библиотеки *Boost.Align*, основным участником поддержки библиотек *Boost.SmartPointers* и *Boost Core*, а также внес вклад в ряд других библиотек Boost. Участвует в поддержке стандарта C++, создавая документы по предложениям и отчеты о дефектах, и у него даже есть по крайней мере одна функция, принятая для будущего стандарта C++20 (P0674r1: расширение make_shared для поддержки массивов). Глен живет со своей женой Кэролайн и дочерью Айрин в США. Он окончил Университет Сиднея в Австралии, до этого жил в Новой Зеландии.

Марков Кирилл увлёкся программированием ещё в школе. Начал заниматься коммерческой разработкой ПО с ранних курсов университета. С тех пор освоил множество платформ, технологий и языков программирования, является full stack разработчиком, но предпочитает backend разработку. На данный момент живёт и трудится в Москве ведущим программистом в одной из крупнейших компаний. Проповедует педантичный формализованный подход к процессам разработки. Неисправимый любитель чая и интересной беседы.

Вступительное слово автора

Более 10 лет назад, когда я только начал осваивать C++, с хорошей литературой было очень тяжко. В итоге навыки C++ приходилось оттачивать, изучая исходные коды библиотеки Boost. Дело двигалось очень медленно, все было непонятно, документация и комментарии на английском не сильно помогали. С тех пор прошло уже много лет, библиотеки Boost отчасти стали стандартом C++ и продолжают развиваться, опережая по своим возможностям стандартную библиотеку C++ на десятки лет. Функционал внутри Boost огромен… и все так же непонятен для начинающих.
Эта книга содержит ответы на типичные вопросы:

- Как мне решить вот эту проблему?
- Как это работает?
- Как это устроено под капотом?
- Как бы поэкспериментировать, не заморачиваясь с настройкой окружения?
- А разве подобного нет в стандартной библиотеке?
- А какие есть хитрости при работе с этим инструментом?
- А есть ли способ решить это получше?
- А что еще почитать на эту тему?

Другими словами, эта книга – тот помощник, которого мне не хватало в свое время и которого не хватает многим разработчикам поныне.

Надеюсь, вам понравится!

Антон Полухин,
представитель России в Международном комитете по стандартизации C++,
разработчик и автор многих библиотек Boost,
руководитель группы Общих Компонент в Яндекс.Такси,
сопредседатель РГ21 C++ и модератор https://stdcpp.ru ,
спикер на конференциях РГ21, Corehard, C++ Russia,
корпоративный консультант по вопросам C++ https://apolukhin.github.io/ ,
автор этой книги :)

Вступительное слово от сообщества C++

История собрания библиотек Boost насчитывает почти столько же лет, сколько и стандарт языка C++. Это означает, что сообщество программистов с первых лет остро ощущало нехватку в стандарте необходимых инструментов и брало инициативу в свои руки. Ничего удивительного, что уже через несколько лет Boost прочно занял место неофициального стандарта и заодно испытательного полигона для смелых идей – кандидатов на включение в будущие версии стандарта.

Кроме того, Boost можно считать своеобразным эталоном качества библиотечного кода. Реализации структур данных и алгоритмов имеют под собой солидное математическое обоснование (скажем, где были бы контейнеры Boost.MultiIndex без строгого понятия точной нижней или верхней грани, а алгоритмы – без строгой меры временной сложности, знаменитого O-большого). С другой стороны, основываются на филигранной технике использования всех средств языка (чего стоит одно метапрограммирование на шаблонах). И все это умножается на необходимость поддерживать различные платформы, разные стандарты языка C++ и требование сохранять концептуальную согласованность между разными библиотеками. Круг задач, для которых в Boost можно найти готовое решение, поражает воображение.

Однако широчайший охват задач вместе с высочайшим качеством достается не бесплатно. Чтобы отыскать в Boost наиболее подходящий инструмент и гибко настроить на свою конкретную задачу, от программиста требуется определенный уровень профессиональной культуры и широта кругозора. Поэтому если в отсутствие Boost программист-профессионал страдает от нехватки необходимых средств, то при наличии Boost новичок может страдать от необъятной груды функций и классов без надежды в ней разобраться. Если первой эмоцией от знакомства с собранием библиотек Boost обычно бывает восхищение от ее мощи, стройности и продуманности всех мелочей, то второй – испуг: как все это постичь и не запутаться и где добыть книгу-путеводитель от простого к сложному. Наконец, когда благодаря Антону Полухину такая книга появилась, остается чистое восхищение.

Boost, равно как и STL, в силу необходимости быть кросс-платформенным, поддерживать множество опций компилятора и огромное количество взаимосвязей внутри самой библиотеки, выглядит весьма громоздким и сложным для понимания. В своей книге Антон Полухин рассматривает не только применение элементов библиотеки, но и указывает на особенности их реализации, что невозможно переоценить: понимание подкапотных процессов позволяет осознанно использовать элементы библиотеки с точки зрения корректности и производительности кода, а при необходимости и заменить их на аналогичные собственного производства меньшей кровью. Из вышеперечисленного естественным образом следует и просветительская функция данной книги, поскольку внимательный читатель может почерпнуть из нее как новые идеи, так и освежить особенности различных стандартов языка.

C++ сообщество, CoreHard 2020

Предисловие

Если вы хотите воспользоваться преимуществами Boost и C++ и не путаться, какую библиотеку в какой ситуации использовать, тогда эта книга для вас. Начиная с основ, вы перейдете к изучению того, как библиотеки Boost упрощают разработку приложений. Вы научитесь преобразовывать данные: строки в числа, числа в строки, числа в числа и многое другое. Управление ресурсами станет проще некуда. Вы увидите, какую работу можно выполнить во время компиляции и на что способны контейнеры Boost. Вы узнаете все, что нужно, для разработки качественных, быстрых и портативных приложений. Напишите программу один раз, и вы сможете использовать ее в операционных системах Linux, Windows, macOS и Android. От манипулирования изображениями до графов, каталогов, таймеров, файлов и работы в сети – каждый найдет для себя интересную тему.

Обратите внимание, что знания, полученные в ходе прочтения этой книги, не устареют, поскольку все больше и больше библиотек Boost становятся частью стандарта C++.

О чем эта книга

Глава 1 «*Приступаем к написанию приложения*» рассказывает о библиотеках для повседневного использования. Мы увидим, как получить параметры конфигурации из разных источников и как упростить себе жизнь, используя некоторые из типов данных, представленных авторами библиотеки Boost.

Глава 2 «*Управление ресурсами*» посвящена типам данных, представленных библиотеками Boost, по большей части фокусируясь на работе с указателями. Мы увидим, как с легкостью управлять ресурсами и использовать тип данных, способный хранить любые функциональные объекты, функции и лямбда-выражения. После прочтения этой главы ваш код станет более надежным, а утечки памяти уйдут в прошлое.

Глава 3 «*Преобразование и приведение*» описывает, как преобразовывать строки, числа и пользовательские типы друг в друга, как безопасно приводить полиморфные типы и как писать маленькие и большие парсеры прямо в исходных файлах C++. Рассматривается несколько способов преобразования данных как для повседневного использования, так и для редких случаев.

Глава 4 «*Уловки времени компиляции*» описывает базовые приемы библиотек Boost, которые можно использовать при проверках во время компиляции, для настройки алгоритмов и в других задачах метапрограммирования. Понимание исходных файлов Boost и других схожих с Boost библиотек без этого невозможно.

Глава 5 «*Многопоточность*» посвящена основам многопоточного программирования и синхронизации доступа к данным.

Глава 6 «*Манипулирование задачами*» показывает, как работать с функциональными объектами – задачами. Основная идея этой главы заключается

в том, что мы можем разделить всю обработку, вычисления и взаимодействия на функторы (задачи) и обрабатывать каждую из этих задач практически независимо. Более того, мы можем не блокировать поток выполнения на некоторых медленных операциях (таких как получение данных из сокета или ожидание тайм-аута), а вместо этого предоставить задачу обратного вызова (callback) и продолжить работу с другими задачами. Как только ОС закончит медленную операцию, будет выполнен наш обратный вызов.

Глава 7 «*Манипулирование строками*» показывает различные аспекты изменения, поиска и представления строк. Мы увидим, как можно с легкостью выполнять некоторые распространенные задачи, связанные со строками, с помощью библиотек Boost.

Глава 8 «*Метапрограммирование*» раскрывает классные и трудные для понимания методы программирования на этапе компиляции. В этой главе мы посмотрим, как можно упаковать несколько типов в один тип, подобный кортежу. Мы создадим функции для управления коллекциями типов, посмотрим, как можно изменять типы коллекций во время компиляции и как трюки во время компиляции можно смешивать с вычислениями времени выполнения (runtime).

Глава 9 «*Контейнеры*» рассказывает о boost-контейнерах и вещах, непосредственно связанных с ними. В этой главе содержится информация о классах Boost, которые можно использовать в повседневном программировании, что сделает ваш код намного быстрее и облегчит разработку новых приложений.

Глава 10 «*Сбор информации о платформе и компиляторе*» описывает различные вспомогательные макросы, используемые для обнаружения возможностей компилятора, платформы и функциональности Boost. Вы познакомитесь с макросами, которые широко используются в библиотеках Boost и которые необходимы для написания переносимого кода, способного работать с любыми флагами компилятора.

Глава 11 «*Работа с системой*» подробно рассматривает файловую систему и способы создания и удаления файлов. Мы увидим, как данные могут передаваться между различными системными процессами, как читать файлы с максимальной скоростью и как решать другие системные задачи.

Глава 12 «В*ерхушка айсберга*» посвящена большим библиотекам Boost и знакомит вас с необходимыми для их использования основами.

Что нужно для этой книги

Вам нужен современный компилятор C++, библиотеки Boost (подойдет любая версия, но рекомендуется 1.65 или более новая) и среда разработки QtCreator и утилита qmake. Или просто перейдите по адресу http://apolukhin.GitHub.io/Boost-Cookbook, чтобы запускать примеры и экспериментировать с ними в режиме онлайн.

Для кого эта книга

Эта книга предназначена для разработчиков, желающих улучшить свои знания в области Boost и упростить процессы разработки приложений. Предполагается, что вы уже знакомы с C++ и имеете базовые знания стандартной библиотеки.

Разделы

В этой книге вы найдете несколько заголовков, которые часто появляются в тексте («Подготовка», «Как это делается…», «Как это работает…», «Дополнительно…» и «См. также»). Чтобы предоставить четкие инструкции относительно того, как завершить рецепт, мы будем использовать эти разделы следующим образом:

Подготовка

В этом разделе рассказывается, чего ожидать в рецепте, и описывается, как настроить какое-либо программное обеспечение или какие-либо предварительные параметры, необходимые для рецепта.

Как это делается…

Этот раздел содержит шаги, необходимые для того, чтобы следовать рецепту.

Как это работает…

Данный раздел обычно состоит из подробного объяснения того, что произошло в предыдущем разделе.

Дополнительно…

Этот раздел состоит из дополнительной информации о рецепте, чтобы читатель был более осведомлен.

См. также

Данный раздел содержит полезные ссылки на другую полезную информацию для рецепта.

Обозначения

В этой книге вы найдете ряд текстовых стилей, используемых для того, чтобы различать разные виды информации. Вот несколько примеров этих стилей и объяснение их значения.

Код в тексте, имена таблиц базы данных, имена папок, имена файлов, расширения файлов, пути, фиктивные URL-адреса, пользовательский ввод и учетные записи в Twitter отображаются следующим образом:

«Помните, что эта библиотека состоит не только из заголовочных файлов, ваша программа должна линковаться с `libboost_program_options`.».

Блок кода выглядит так:

```
#include <boost/program_options.hpp>
#include <iostream>
namespace opt = boost::program_options;
int main(int argc, char *argv[])
{
```

Когда мы хотим обратить ваше внимание на определенную часть блока кода, соответствующие строки или элементы выделяются жирным шрифтом:

```
#include <boost/program_options.hpp>
#include <iostream>
namespace opt = boost::program_options;
int main(int argc, char *argv[])
```

Любой ввод или вывод командной строки записывается следующим образом:

```
  $ ./our_program.exe --apples=10 --oranges=20
Fruits count: 30
```

Новые термины и **важные слова** выделены жирным шрифтом.

Предупреждения или важные заметки сопровождаются таким знаком.

Советы и хитрости выглядят так.

Загрузка примеров кода

Файлы исходных кодов примеров, представленных в этой книге рецептов, есть в репозитории автора на сайте GitHub. Для получения последней версии кода вы можете зайти на страницу https://GitHub.com/apolukhin/Boost-Cookbook.

Глава 1

Приступаем к написанию приложения

Темы, которые мы рассмотрим в этой главе:

- получение параметров конфигурации;
- хранение любого значения в контейнере или переменной;
- хранение одного из нескольких выбранных типов в контейнере или переменной;
- использование более безопасного способа работы с контейнером, в котором хранится один из нескольких выбранных типов;
- возвращение значения или флага «значения нет»;
- возвращение массива из функции;
- объединение нескольких значений в одно;
- привязка и переупорядочение параметров функции;
- получение удобочитаемого имени типа;
- использование эмуляции перемещения C++11;
- создание некопируемого класса;
- создание некопируемого, но перемещаемого класса;
- использование алгоритмов C++14 и C++11.

Вступление

Boost – это коллекция библиотек для языка C++. Каждая из этих библиотек была проверена множеством профессиональных программистов, прежде чем была принята Boost. Библиотеки тестируются на многих платформах с использованием множества компиляторов и реализаций стандартной библиотеки C++. Используя Boost, вы можете быть уверены, что в ваших руках находится одно из самых портативных, быстрых и надежных решений, которое распространяется по лицензии, подходящей для коммерческих проектов и проектов с открытым исходным кодом.

Многие части Boost были включены в стандарты C++11, C++14, C++17 и C++20. Кроме того, некоторые библиотеки Boost попадут и в следующие стандарты C++. В каждом рецепте в этой книге вы найдете примечания, касающиеся стандарта C++.

Без долгого вступления, давайте начнем!

В этой главе мы увидим несколько рецептов для повседневного использования. Мы увидим, как получить параметры конфигурации из разных источников и что можно сделать, используя популярные типы данных, представленные авторами библиотек Boost.

Получение параметров конфигурации

Взгляните на консольные программы, такие как `cp` в Linux. Все они имеют красивую «справку», их входные параметры не зависят от какой-либо позиции и имеют читабельный синтаксис. Например:

```
$ cp --help
Usage: cp [OPTION]... [-T] SOURCE DEST
  -a, --archive           same as -dR --preserve=all
  -b                      like --backup but does not accept an argument
```

Вы можете реализовать ту же функциональность для своей программы за 10 минут. Все, что вам нужно, – это библиотека `Boost.ProgramOptions`.

Подготовка

Все, что требуется для этого рецепта, – базовые знания C++. Помните, что библиотека `Boost.ProgramOptions` состоит не только из заголовочных файлов, поэтому ваша программа должна линковаться с библиотекой `libboost_program_options`.

Как это делается...

Давайте начнем с простой программы, которая принимает количество яблок (`apples`) и апельсинов (`oranges`) в качестве входных данных и подсчитывает общее количество фруктов. Вот результат, который мы хотим получить:

```
$ ./our_program.exe --apples=10 --oranges=20
Fruits count: 30
```

Выполните следующие шаги.

1. Включите в код заголовочный файл `boost/program_options.hpp` и создайте псевдоним для пространства имен для `boost::program_options` (его слишком долго набирать на клавиатуре!). Нам также понадобится заголовочный файл `<iostream>`:

   ```
   #include <boost/program_options.hpp>
   #include <iostream>
   namespace opt = boost::program_options;
   ```

2. Теперь мы готовы описать наши опции в функции `main()`:

   ```
   int main(int argc, char *argv[])
   {
     // Создаем опции, описывающие переменную, и даем ей текстовое описание
     // «All options».
     opt::options_description desc("All options");

     // Когда мы добавляем опции, первый параметр - это имя, которое будет использоваться
     // в командной строке. Второй параметр - это тип данных опции, заключенный в класс
     // value <>. Третий параметр должен быть кратким описанием этой опции.
   ```

```
desc.add_options()
   ("apples", opt::value<int>(), "how many apples do you have")
   ("oranges", opt::value<int>(), "how many oranges do you have")
   ("help", "produce help message")
;
```

3. Давайте выполним парсинг командной строки:

```
// Переменная для хранения аргументов нашей командной строки
opt::variables_map vm;

// Парсинг и сохранение аргументов
opt::store(opt::parse_command_line(argc, argv, desc), vm);

// Эта функция должна вызываться после парсинга и сохранения.
opt::notify(vm);
```

4. Добавим немного кода для обработки опции `help`:

```
if (vm.count("help")) {
    std::cout << desc << "\n";
    return 1;
}
```

5. Заключительный этап. Подсчет фруктов можно реализовать следующим образом:

```
    std::cout << "Fruits count: "
         << vm["apples"].as<int>() + vm["oranges"].as<int>()
         << std::endl;
} // Конец функции `main`
```

Теперь если мы вызовем нашу программу, используя параметр `help`, то получим следующий вывод:

```
All options:
    --apples arg       how many apples do you have
    --oranges arg      how many oranges do you have
    --help             produce help message
```

Как видите, в коде мы не предоставляем тип данных для значения параметра `help`, потому что не ждем, что ему будут переданы какие-либо значения.

Как это работает...

Этот пример довольно просто понять, исходя из кода и комментариев. Его запуск дает ожидаемый результат:

```
  $ ./our_program.exe --apples=100 --oranges=20
Fruits count: 120
```

Дополнительно...

Стандарт C++ принял множество библиотек Boost; однако вы не найдете `Boost.ProgramOptions` даже в C++20. В настоящее время ее не планируют включать и в C++23.

Библиотека `ProgramOptions` очень мощная и имеет множество возможностей. Она может:

- записать значения параметров конфигурации непосредственно в переменную и сделать этот параметр обязательным:

```
int oranges_var = 0;
desc.add_options()
    // ProgramOptions сохраняет значение параметра в переменную, которая передается
    // по указателю. Здесь значение параметра «--oranges» будет сохранено в «oranges_var».
    ("oranges,o", opt::value<int>(&oranges_var)->required(),
                                        "oranges you have")
```

- получить необязательный строковый параметр :

```
// Опция 'name' не помечена как 'required ()', поэтому пользователь может
// не предоставлять ее.
("name", opt::value<std::string>(), "your name")
```

- добавить сокращенное обозначение и установить 10 в качестве значения по умолчанию для `apples`:

```
// «a» - сокращенное обозначение для яблок. Используйте его как в «-a 10».
// Если значение не указано, используется значение по умолчанию.
("apples,a", opt::value<int>()->default_value(10),
                       "apples that you have");
```

- получить недостающие параметры из файла конфигурации:

```
opt::variables_map vm;

// Парсинг параметров командной строки и сохранение значений в 'vm'.
opt::store(opt::parse_command_line(argc, argv, desc), vm);

// Мы также можем выполнить парсинг переменных окружения. Просто используйте функцию
// 'opt::store' с 'opt::parse_environment'.
// Добавляем недостающие параметры из конфигурационного файла "apples_oranges.cfg".
try {
    opt::store(
        opt::parse_config_file<char>("apples_oranges.cfg", desc),
        vm
    );
} catch (const opt::reading_file& e) {
    std::cout << "Error: " << e.what() << std::endl;
}
```

Синтаксис конфигурационного файла отличается от синтаксиса командной строки. Нам не нужно ставить знаки «минус» перед опциями. Поэтому наш файл `apple_oranges.cfg` должен выглядеть так:

```
oranges=20
```

- проверить, что были установлены все необходимые параметры:

```
try {
    // Если один из обязательных параметров не был задан, выбрасывается исключение
    // `opt::required_option`
    opt::notify(vm);
} catch (const opt::required_option& e) {
```

```
        std::cout << "Error: " << e.what() << std::endl;
        return 2;
    }
```

Если мы объединим все вышеупомянутые советы в один исполняемый файл, то команда `help` выдаст такой вывод:

```
$ ./our_program.exe --help
 All options:
   -o [ --oranges ] arg          oranges that you have
   --name arg                    your name
   -a [ --apples ] arg (=10)     apples that you have
   --help                        produce help message
```

Если запустить его без конфигурационного файла, это приведет к следующему выводу:

```
$ ./our_program.exe
 Error: can not read options configuration file 'apples_oranges.cfg'
 Error: the option '--oranges' is required but missing
```

Запуск программы с `oranges=20` в конфигурационном файле сгенерирует 30, поскольку для яблок по умолчанию установлено значение 10:

```
$ ./our_program.exe
 Fruits count: 30
```

См. также

- Официальная документация по Boost содержит еще много примеров. В ней рассказывается о более продвинутых функциях `Boost.ProgramOptions`, таких как параметры, зависящие от позиции, нетрадиционный синтаксис и т. д.; посетите страницу http://boost.org/libs/program_options;
- вы можете изменять и запускать все примеры из этой книги в режиме онлайн на странице http://apolukhin.github.io/Boost-Cookbook/.

Сохранение любого значения в контейнере или переменной

Если вы программировали на Java, C # или Delphi, то в C++ вам определенно не хватает возможности создания контейнеров с типом значения `ObjectC++`. Класс `Object` в этих языках является базовым классом почти для всех типов, поэтому вы можете в любое время присвоить ему практически любое значение. Только представьте, как было бы здорово иметь такую возможность в C++:

```
typedef std::unique_ptr<Object> object_ptr;

std::vector<object_ptr> some_values;
some_values.push_back(new Object(10));
some_values.push_back(new Object("Hello there"));
some_values.push_back(new Object(std::string("Wow!")));

std::string* p = dynamic_cast<std::string*>(some_values.back().get());
```

```
assert(p);
(*p) += " That is great!\n";
std::cout << *p;
```

Подготовка

Мы будем работать с библиотекой header-only (состоящей только из заголовочных файлов). Все, что требуется для этого рецепта, – базовые знания C++.

Как это делается...

Boost предлагает решение, библиотеку `Boost.Any`, которая имеет даже более выразительный синтаксис:

```
#include <iostream>
#include <vector>
#include <string>

int main() {
    std::vector<boost::any> some_values;
    some_values.push_back(10);
    some_values.push_back("Hello there!");
    some_values.push_back(std::string("Wow!"));

    std::string& s = boost::any_cast<std::string&>(some_values.back());
    s += " That is great!";
    std::cout << s;
}
```

Здорово, правда? Кстати, у `boost::any` есть пустое состояние, которое можно проверить с помощью функции-члена `empty()` (как в контейнерах стандартных библиотек).

Можно получить значение из `boost::any`, используя два подхода:

```
void example() {
    boost::any variable(std::string("Hello world!"));

    // При использовании приведенного ниже метода может выбрасываться исключение
    // boost::bad_any_cast, если фактическое значение в переменной
    // не является std::string.
    std::string s1 = boost::any_cast<std::string>(variable);

    // Исключение не будет выброшено. Если фактическое значение в переменной не является
    // std::string, будет возвращен указатель NULL.
    std::string* s2 = boost::any_cast<std::string>(&variable);
}
```

Как это работает...

Класс `boost::any` хранит в себе любое значение. Чтобы добиться этого, он использует метод **стирания типов** (близкий к тому, что Java или C # делает со всеми типами). Чтобы использовать эту библиотеку, вам не нужно подробно знать ее внутреннюю реализацию, но ниже приводится краткий обзор вышеупомянутого метода для любопытных.

При присвоении некоторой переменной типа T библиотека Boost.Any создает экземпляр holder<T>, который может хранить значение указанного типа T и который является производным от некого базового типа placeholder:

```
template<typename ValueType>
struct holder : public placeholder {
    virtual const std::type_info& type() const {
         return typeid(ValueType);
    }
    ValueType held;
};
```

Тип placeholder имеет виртуальные функции для получения std::type_info хранимого типа T и клонирования:

```
struct placeholder {
    virtual ~placeholder() {}
    virtual const std::type_info& type() const = 0;
};
```

Класс boost::any хранит указатель на placeholder. При использовании any_cast<T>() boost::any проверяет, что вызов ptr->type() дает std::type_info, эквивалентный typeid(T), и возвращает static_cast<holder<T>*>(ptr)->held.

Дополнительно...

Такая гибкость не обходится без затрат. Конструирование копий, конструирование значений, копирование присваивания и присваивание значений экземплярам boost::any выполняют динамическое выделение памяти; все приведения типов производят проверки **динамической идентификации типа данных (RTTI)**; класс boost::any часто использует виртуальные функции. Если вас интересует производительность, следующий рецепт даст вам представление о том, как достичь почти таких же результатов без динамических аллокаций и применения RTTI.

Класс boost::any использует **rvalue-ссылки**, но не может использоваться в **constexpr**-функциях.

Библиотека Boost.Any была принята в стандарт C++17. Если ваш компилятор совместим с C++17 и вы хотите избежать использования Boost для any, просто замените пространство имен boost на пространство имен std и подключите заголовочный файл <any> вместо <boost/any.hpp>. Ваша стандартная реализация библиотеки может работать немного быстрее, если вы храните крошечные объекты в std::any.

У std::any *есть функция* reset() *вместо функции* clear() *и* has_value() *вместо* empty(). *Почти все исключения в Boost наследуются от класса* std::exception *или из его производных, например* boost::bad_any_cast *является производным от* std::bad_cast. *Это означает, что вы можете перехватывать почти все исключения Boost с помощью* catch (const std::exception& e).

См. также

- В официальной документации по Boost можно найти еще несколько примеров; посетите страницу http://boost.org/libs/any.
- Рецепт «*Использование более безопасного способа работы с контейнером, в котором хранится несколько выбранных типов*» приводится для получения дополнительной информации по теме.

Хранение одного из нескольких выбранных типов в контейнере или переменной

Объединения (union) C++03 могут содержать только очень простые типы под названием **простая структура данных (POD)**. Например, в C++03 нельзя хранить `std::string` или `std::vector` в объединении.

Вы знаете о концепции **неограниченных объединений (unrestricted unions)** в C++11? Позвольте мне кратко рассказать вам о них. C++11 ослабляет требования для объединений, но вы должны сами управлять созданием и уничтожением не-POD-типов. Вы должны вызывать конструирование или уничтожение по месту (in-place contrcution/destruction) и запомнить, какой тип хранится в объединении. Огромный объем работы, не так ли?

Можно ли в C++03 получить переменную, которая ведет себя как неограниченное объединение C++ и которая управляет временем жизни объекта, запоминает его тип?

Подготовка...

Мы будем работать с библиотекой header-only, которая проста в использовании. Все, что требуется для этого рецепта, – базовые знания C++.

Как это делается...

Позвольте представить вам библиотеку `Boost.Variant`.

1. Библиотека `Boost.Variant` может хранить любые типы, указанные во время компиляции. Она также управляет созданием или уничтожением по месту, и ей даже не требуется стандарт C++11:

```
#include <boost/variant.hpp>
#include <iostream>
#include <vector>
#include <string>

int main() {
    typedef boost::variant<int, const char*, std::string> my_var_t;
    std::vector<my_var_t> some_values;
    some_values.push_back(10);
    some_values.push_back("Hello there!");
    some_values.push_back(std::string("Wow!"));

    std::string& s = boost::get<std::string>(some_values.back());
    s += " That is great!\n";
```

```
        std::cout << s;
    }
```

Здорово, правда?

2. Boost.Variant не имеет пустого состояния, но у нее есть функция empty(), которая бесполезна и всегда возвращает значение false. Если вам нужно представить пустое состояние, просто добавьте простой тип первым шаблонным параметром boost::variant. Если Boost.Variant содержит этот тип, интерпретируйте его как пустое состояние. Вот пример, в котором мы будем использовать тип boost::blank для представления пустого состояния:

```
void example1() {
  // Конструктор по умолчанию создает экземпляр boost::blank.
  boost::variant<
      boost::blank, int, const char*, std::string
  > var;
  // Метод which() возвращает индекс типа, который в настоящее время содержится
  // в  variant
  assert(var.which() == 0); // boost::blank

  var = "Hello, dear reader";
  assert(var.which() != 0);
}
```

3. Можно получить значение из boost::variant, используя два подхода:

```
void example2() {
  boost::variant<int, std::string> variable(0);

  // При использовании приведенного ниже метода может выбрасываться исключение
  // boost::bad_get, если фактическое значение в  variable не является int.
  int s1 = boost::get<int>(variable);

  // Если фактическое значение в переменной не является int, будет возвращено NULL.
  int* s2 = boost::get<int>(&variable);
}
```

Как это работает...

Класс boost::variant содержит массив байтов и хранит значения в этом массиве. Размер массива определяется во время компиляции путем применения функции sizeof() и функций для определения выравнивания (alignment) каждого из типов шаблонов. При присваивании или создании класса boost::variable предыдущее значение уничтожается по месту, а новое значение создается поверх массива байтов с использованием оператора placement new.

Дополнительно...

Boost.Variant обычно не выделяет память динамически и не требует RTTI. Это чрезвычайно быстрая библиотека, и она широко используется другими библиотеками Boost. Для достижения максимальной производительности убедитесь, что в шаблонном списке типов в первой позиции указан простой

тип (POD). boost::variant использует rvalue-ссылки C++11, если они доступны в вашем компиляторе.

Библиотека Boost.Variant является частью стандарта C++17. std::variant немного отличается от boost::variant:

- std::variant объявляется в файле заголовка <variant>, а не в <boost/variant.hpp>;
- std::variant никогда динамически не выделяет память;
- std::variant можно использовать в constexpr-функциях;
- вместо того чтобы писать boost::get<int>(& variable), вы должны писать std::get_if<int> (&variable);
- std::variant не может рекурсивно содержать сам себя и лишен некоторых других передовых методик;
- std::variant имеет конструкторы in-place;
- std::variant имеет функцию index() вместо which().

См. также

- Рецепт «*Использование безопасного способа работы с контейнером, в котором хранится несколько выбранных типов*»;
- в официальной документации Boost содержатся дополнительные примеры и описания некоторых других функций библиотеки Boost.Variant. Их можно найти по адресу: http://boost.org/libs/variant;
- поэкспериментируйте с кодом в режиме онлайн на странице http://apolukhin.github.io/Boost-Cookbook.

Использование более безопасного способа работы с контейнером, в котором хранится один из нескольких выбранных типов

Представьте, что вы создаете C++-обертку для некоторого низкоуровневого интерфейса базы данных SQL. Вы решили, что класс boost::any будет идеально соответствовать требованиям одной ячейки таблицы базы данных.

Какой-то другой программист будет использовать ваши классы, и его/ее задачей будет получить строку из базы данных и посчитать сумму арифметических типов в строке.

Вот как будет выглядеть такой код:

```
#include <boost/any.hpp>
#include <vector>
#include <string>
#include <typeinfo>
#include <algorithm>

#include <iostream>
// typedefы и методы будут в нашем заголовке, который оборачивает
// нативный интерфейс SQL.
typedef boost::any cell_t;
```

```
typedef std::vector<cell_t> db_row_t;
// Это всего лишь пример, реальной работы с базой данных нет.
db_row_t get_row(const char* /*query*/) {
    // В реальном приложении параметр 'query' должен иметь тип 'const char *'
    // или 'const std::string &'? См. рецепт "Использование типа «ссылка на строку»",
    // чтобы найти ответ.
    db_row_t row;
    row.push_back(10);
    row.push_back(10.1f);
    row.push_back(std::string("hello again"));
    return row;
}

// Так пользователь будет использовать ваши классы обертки
struct db_sum {
private:
    double& sum_;
public:
    explicit db_sum(double& sum)
        : sum_(sum)
    {}

     void operator()(const cell_t& value) {
         const std::type_info& ti = value.type();
         if (ti == typeid(int)) {
             sum_ += boost::any_cast<int>(value);
         } else if (ti == typeid(float)) {
              sum_ += boost::any_cast<float>(value);
         }
       }
};

int main() {
    db_row_t row = get_row("Query: Give me some row, please.");
    double res = 0.0;
    std::for_each(row.begin(), row.end(), db_sum(res));
    std::cout << "Sum of arithmetic types in database row is: "
              << res << std::endl;
}
```

Если вы скомпилируете и запустите этот пример, он выдаст правильный ответ:

```
Sum of arithmetic types in database row is: 20.1
```

Вы помните, о чем думали, когда читали реализацию `operator()`? Полагаю, вы думали так: «*А как насчет* `double`, `long double`, `short`, `unsigned` *и других типов*?» Те же мысли придут в голову программисту, который будет использовать ваш интерфейс. Поэтому вам нужно тщательно документировать значения, хранящиеся в вашем типе `cell_t`, или использовать более элегантное решение, которое описано в следующих разделах.

Подготовка

Если вы еще незнакомы с библиотеками `Boost.Variant` и `Boost.Any`, настоятельно рекомендую вам прочитать два предыдущих рецепта.

Как это делается...

Библиотека `Boost.Variant` реализует идиому проектирования «Посетитель» (visitor) для доступа к хранимым данным. Это намного безопаснее, чем получать значения с помощью `boost::get<>`. Данная идиома заставляет программиста заботиться о каждом типе в `variant`, иначе код нельзя будет скомпилировать. Вы можете воспользоваться этой идиомой с помощью функции `boost::apply_visitor`, которая принимает функциональный объект `visitor` в качестве первого параметра и `variant` в качестве второго параметра. Если вы используете компилятор стандарта, предшествующего C++14, то функциональные объекты `visitor` должны наследоваться от класса `boost::static_visitor<T>`, где `T` – это тип, возвращаемый `visitor`. У объекта `visitor` должны быть перегрузки `operator()` для каждого типа, который может содержать variant.

Давайте изменим тип `cell_t` на `boost::variable<int, float, string>` и изменим наш пример:

```
#include <boost/variant.hpp>
#include <vector>
#include <string>
#include <iostream>

// typedefы и методы будут в нашем заголовке, который оборачивает интерфейс SQL.
typedef boost::variant<int, float, std::string> cell_t;
typedef std::vector<cell_t> db_row_t;

// Это всего лишь пример, реальной работы с базой данных нет.
db_row_t get_row(const char* /*query*/) {
    // См. рецепт "Использование типа «ссылка на строку»",
    // где приводится более подходящий тип для параметра "query".
    db_row_t row;
    row.push_back(10);
    row.push_back(10.1f);
    row.push_back("hello again");
    return row;
}

// Это код, необходимый для суммирования значений.
// Мы могли бы просто использовать boost::static_visitor<>,
// если бы наш объект visitor ничего не возвращал.
struct db_sum_visitor: public boost::static_visitor<double> {
    double operator()(int value) const {
        return value;
    }
    double operator()(float value) const {
        return value;
    }
```

```
        double operator()(const std::string& /*value*/) const {
            return 0.0;
        }
};

int main() {
    db_row_t row = get_row("Query: Give me some row, please.");
    double res = 0.0;
    for (auto it = row.begin(), end = row.end(); it != end; ++it) {
        res += boost::apply_visitor(db_sum_visitor(), *it);
    }
    std::cout << "Sum of arithmetic types in database row is: "
              << res << std::endl;
}
```

Как это работает...

Во время компиляции библиотека Boost.Variant генерирует большой оператор switch, каждый case которого вызывает объект visitor для одного типа из списка типов варианта. Во время выполнения индекс сохраненного типа извлекается с помощью функции which(), и происходит переход на нужный case в switch. Нечто подобное будет сгенерировано для boost::variant<int, float, std::string>:

```
switch (which())
{
case 0 /*int*/:
    return visitor(*reinterpret_cast<int*>(address()));
case 1 /*float*/:
    return visitor(*reinterpret_cast<float*>(address()));
case 2 /*std::string*/:
    return visitor(*reinterpret_cast<std::string*>(address()));
default: assert(false);
}
```

Здесь функция address() возвращает указатель на внутреннюю память boost::variant<int, float, std::string>.

Дополнительно...

Если мы сравним этот пример с первым примером из данного рецепта, то увидим следующие преимущества boost::variant:

- мы знаем, какие типы может хранить переменная;
- если разработчик библиотеки интерфейса SQL добавляет или изменяет тип, содержащийся в variant, вместо неправильного поведения мы получим ошибку времени компиляции.

std::variant из стандарта C++17 также поддерживает эту идиому проектирования. Просто напишите std::visit вместо boost::apply_visitor – и готово.

Вы можете скачать примеры файлов с кодами для всех книг Packt, которые вы приобрели, со своего аккаунта на сайте http://www.PacktPub.com. Если вы купили эту книгу в другом месте, можете зайти на страницу http://www.PacktPub.com/support и зарегистрироваться, чтобы получить файлы по электронной почте.

См. также

- Прочитав рецепты из главы 4 «Уловки времени компиляции», вы сможете создавать универсальные объекты visitor, которые будут работать правильно, даже при изменении базовых типов;
- официальная документация Boost содержит дополнительные примеры и описание некоторых других функций библиотеки Boost.Variant: http://boost.org/libs/variant.

Возврат значения или флага «значения нет»

Представьте, что у нас есть функция, которая не выбрасывает исключение, а возвращает значение или как-то говорит о том, что произошла ошибка. В языках программирования Java или C# такие случаи обрабатываются путем сравнения возвращаемого значения из функции с нулевым указателем. Если функция вернула ноль, то произошла ошибка. В языке C++ возврат указателя из функции сбивает с толку пользователей библиотеки и обычно требует медленного динамического выделения памяти.

Подготовка

Все, что вам нужно для этого рецепта, – базовые знания C++.

Как это делается...

Дамы и господа, позвольте мне представить вам библиотеку `Boost.Optional`, используя приведенный ниже пример.

Есть некая функция `try_lock_device()`. Она пытается захватить уникальный доступ к устройству и вернуть объект, владеющий устройством. Операция может успешно завершиться или провалиться, в зависимости от различных условий (в нашем примере – от вызова функции `try_lock_device_impl()`):

```
#include <boost/optional.hpp>
#include <iostream>

class locked_device {
    explicit locked_device(const char* /*param*/) {
       // У нас есть уникальный доступ к устройству.
       std::cout << "Device is locked\n";
    }
    static bool try_lock_device_impl();
public:
    void use() {
        std::cout << "Success!\n";
    }
```

```
    static boost::optional<locked_device> try_lock_device() {
      if (!try_lock_device_impl()) {
        // Не удалось получить уникальный доступ.
        return boost::none;
      }

      // Успешно!
      return locked_device("device name");
    }

    ~locked_device(); // Снимает блокировку с устройства.
};
```

В примере функция возвращает переменную `boost::optional`, которую можно преобразовать в тип `bool`. Так, в нашем примере если возвращаемое значение преобразуется в `true`, тогда мы захватили устройство, и экземпляр класса для работы с устройством можно получить путем разыменования возвращаемой `boost::optional` переменной:

```
int main() {
    for (unsigned i = 0; i < 10; ++i) {
        boost::optional<locked_device> t
             = locked_device::try_lock_device();

        // Можно преобразовать в bool.
        if (t) {
           t->use();
           return 0;
        } else {
            std::cout << "...trying again\n";
        }
    }

    std::cout << "Failure!\n";
    return -1;
}
```

Эта программа выведет следующее:

```
...trying again
...trying again
Device is locked
Success!
```

Созданную по умолчанию переменную `optional` можно преобразовать в `false`, и она не должна быть разыменована, потому что у нее нет базового типа.

Как это работает...

`boost::optional<T>` «под капотом» имеет правильно выровненный массив байтов, в котором объект типа `T` может быть создан по месту, а также переменную `bool` для запоминания состояния объекта `T` (сконструирован он или нет?).

Дополнительно...

Класс `Boost.Optional` не использует динамическую аллокацию памяти, и ему не требуется конструктор по умолчанию для базового типа. Текущая реализация `boost::optional` может работать с rvalue-ссылками в C++11, но ее нельзя использовать в constexpr-функциях.

Если у вас есть класс T, у которого нет пустого состояния, но ваша программная логика требует пустого состояния или неинициализированного T, нужно как-то выкручиваться. Зачастую пользователи применяют умный указатель для класса T, используют нулевой указатель для обозначения пустого состояния и динамически выделяют T, если пустое состояние не требуется. Не делайте этого! Вместо этого используйте boost::optional<T>. Это гораздо более быстрое и надежное решение.

Стандарт C++17 включает в себя класс `std::optional`. Просто замените `<boost/optional.hpp>` на `<optional>` и `boost::` на `std::`, чтобы использовать стандартную версию этого класса. Класс `std::optional` подходит для использования в constexpr-функциях.

См. также

В официальной документации по Boost содержатся дополнительные примеры и описываются расширенные функции `Boost.Optional` (например, конструкторы in-place). Документация доступна по адресу: http://boost.org/libs/optional.

Возвращение массива из функции

Давайте поиграем в угадайку! Что можно сказать об этой функции?

```
char* vector_advance(char* val);
```

Должно возвращаемое значение быть освобождено (деаллоцировано) программистом или нет? Функция пытается освободить входной параметр? Должен ли входной параметр заканчиваться нулем или функция должна предполагать, что этот параметр имеет какую-то определенную длину?

Теперь усложним задачу! Взгляните на следующую строку:

```
char ( &vector_advance( char (&val)[4] ) )[4];
```

Не волнуйтесь. Я также полчаса чесал голову, прежде чем понял, что здесь происходит. `vector_advance` – это функция, которая принимает и возвращает массив из четырех элементов. Можно ли написать эту функцию так, чтобы было понятно?

Подготовка

Все, что вам нужно для этого рецепта, – базовые знания C++.

Как это делается...

Мы можем переписать функцию следующим образом:

```
#include <boost/array.hpp>

typedef boost::array<char, 4> array4_t;
array4_t& vector_advance(array4_t& val);
```

Здесь boost::array<char, 4> – это простая обертка вокруг массива из четырех элементов char.

Этот код отвечает на все вопросы из нашего первого примера и является гораздо более читабельным, чем код из второго примера.

Как это работает...

boost::array – это массив фиксированного размера. Первый шаблонный параметр boost::array является типом элемента, а второй – это размер массива. Если вам нужно изменить размер массива во время выполнения, то класс boost::array вам не подходит, используйте std::vector, boost::container::small_vector, boost::container::stack_vector или boost::container::vector.

Класс boost::array<> не имеет рукописных конструкторов, и все его члены являются открытыми, поэтому компилятор будет рассматривать его как простую структуру данных.

Дополнительно...

Давайте посмотрим еще несколько примеров использования boost::array:

```
#include <boost/array.hpp>
#include <algorithm>
typedef boost::array<char, 4> array4_t;

array4_t& vector_advance(array4_t& val) {
    // Лямбда-функция C++11
    const auto inc = [](char& c){ ++c; };

    // У массива boost::array есть begin(), cbegin(), end(), cend(),
    // rbegin(), size(), empty() и другие функции, которые являются общими
    // для стандартных контейнеров
    std::for_each(val.begin(), val.end(), inc);
    return val;
}

int main() {
    // Можно инициализировать boost::array так же, как массив в C++11:
    // array4_t val = {0, 1, 2, 3};
    // но в C++03 требуется дополнительная пара фигурных скобок.
    array4_t val = {{0, 1, 2, 3}};

    array4_t val_res; // У boost::array есть конструктор по умолчанию
    val_res = vector_advance(val); // и оператор присваивания
    assert(val.size() == 4);
    assert(val[0] == 1);
    /*val[4];*/ // Сработает внутренний assert, т.к. максимальный индекс равен 3

    // Мы можем заставить отработать этот assert во время компиляции.
```

```
    // Интересно? См. рецепт «Проверка размеров во время компиляции»
    assert(sizeof(val) == sizeof(char) * array4_t::static_size);
}
```

Одним из самых больших преимуществ `boost::array` является то, что он не выделяет динамически память и обеспечивает точно такую же производительность, что и обычный массив C. Людям из комитета по стандартизации C++ это тоже понравилось, поэтому он был принят в стандарт C++11. Попробуйте подключить заголовочный файл `<array>` и проверьте наличие `std::array`. `std::array` имеет полную поддержку constexpr начиная с C++17.

См. также

- В официальной документации по Boost приводится полный список методов `Boost.Array` с описанием сложности метода. Он доступен по ссылке: http://boost.org/libs/array.
- Тип `boost::array` широко используется в разных рецептах, например *см.* рецепт «*Привязка значения в качестве параметра функции*».

Объединение нескольких значений в одно

Хороший подарок для тех, кому нравится `std::pair`. В Boost есть библиотека под названием `Boost.Tuple`. Она похожа на `std::pair`, но также может работать с тройками типов, четверками и даже более крупными коллекциями типов.

Подготовка

Все, что вам нужно для этого рецепта, – базовые знания C++.

Как это делается...

Выполните следующие шаги, чтобы объединить несколько значений в один кортеж (tuple).

1. Чтобы начать работать с кортежами, вам нужно подключить правильный заголовочный файл и объявить переменную типа `boost::tuple`:

```
#include <boost/tuple/tuple.hpp>
#include <string>

boost::tuple<int, std::string> almost_a_pair(10, "Hello");
boost::tuple<int, float, double, int> quad(10, 1.0f, 10.0, 1);
```

2. Получение определенного значения осуществляется с помощью функции `boost::get<N>()`, где `N` – это индекс необходимого значения:

```
#include <boost/tuple/tuple.hpp>

void sample1() {
    const int i = boost::get<0>(almost_a_pair);
    const std::string& str = boost::get<1>(almost_a_pair);
    const double d = boost::get<2>(quad);
}
```

Функция `boost::get<>` имеет множество перегрузок и широко используется в Boost. Мы уже видели, что ее можно использовать с другими библиотеками, в рецепте «*Хранение нескольких выбранных типов в контейнере или переменной*».

3. Вы можете создавать кортежи, используя функцию `boost::make_tuple()`, которую быстрее писать, потому что вам не нужно явно перечислять шаблонные параметры кортежа:

```
#include <boost/tuple/tuple.hpp>
#include <boost/tuple/tuple_comparison.hpp>
#include <set>

void sample2() {
    // Операторы сравнения кортежей определены в заголовке
    // «boost/tuple/tuple_comparison.hpp»
    // Не забудьте включить его!
    std::set<boost::tuple<int, double, int> > s;
    s.insert(boost::make_tuple(1, 1.0, 2));
    s.insert(boost::make_tuple(2, 10.0, 2));
    s.insert(boost::make_tuple(3, 100.0, 2));

    // Требует C++11
    const auto t = boost::make_tuple(0, -1.0, 2);
    assert(2 == boost::get<2>(t));
    // Мы можем выполнить эту проверку на этапе компиляции. Интересно?
    // См. главу «Трюки времени компиляции»
}
```

4. Еще одна функция, облегчающая жизнь, – это `boost::tie()`. Она работает почти как `make_tuple`, но добавляет неконстантную ссылку для каждого из передаваемых типов. Такой кортеж можно использовать для получения значений в переменные из другого кортежа. Это можно лучше понять из приведенного ниже примера:

```
#include <boost/tuple/tuple.hpp>
#include <cassert>

void sample3() {
    boost::tuple<int, float, double, int> quad(10, 1.0f, 10.0,
1);
    int i;
    float f;
    double d;
    int i2;

    // Передача значений из 'quad' в переменные 'i', 'f', 'd', 'i2'.
    boost::tie(i, f, d, i2) = quad;
    assert(i == 10);
    assert(i2 == 1);
}
```

Как это работает...

Некоторые читатели могут задаться вопросом, зачем нам нужен кортеж, когда мы всегда можем написать собственные структуры с более подходящими именами, например вместо того, чтобы писать `boost::tuple<int, std::string>`, мы можем создать структуру:

```
struct id_name_pair {
    int id;
    std::string name;
};
```

Да, эта структура определенно понятнее, чем `boost::tuple<int, std::string>`. Основное применение библиотеки кортежей – упрощение шаблонного программирования.

Дополнительно...

Кортеж работает так же быстро, как шаблон структуры `std::pair` (он не выделяет память в кучи и не имеет виртуальных функций). Комитет по C++ счел этот класс очень полезным, и тот был включен в стандартную библиотеку. Вы можете найти его в C++11-совместимой реализации в файле заголовка `<tuple>` (не забудьте заменить все пространства имен `boost::` на `std::`).

Стандартная версия кортежа имеет несколько микрооптимизаций и, как правило, обеспечивает немного большую производительность. Тем не менее `std::tuple` не гарантирует порядок конструирования элементов. Поэтому если вам нужен кортеж, который создает свои элементы, начиная с первого, вы должны использовать `boost::tuple`:

```
#include <boost/tuple/tuple.hpp>
#include <iostream>

template <int I>
struct printer {
    printer() { std::cout << I; }
};

int main() {
    // На выходе дает 012
    boost::tuple<printer<0>, printer<1>, printer<2> > t;
}
```

Текущая реализация кортежа Boost не использует variadic templates, не поддерживает rvalue-ссылки и декомпозицию C++17 (structured bindings), и ее нельзя использовать в constexpr-функциях.

См. также

- В официальной документации по Boost содержится больше примеров и информации о производительности и возможностях библиотеки `Boost.Tuple`. Она доступна по ссылке http://www.boost.org/libs/tuple;
- в рецепте «*Преобразование всех элементов кортежа в строку*» из главы 8 «*Метапрограммирование*» показаны продвинутые способы использования кортежей.

Привязка и переупорядочение параметров функции

Если вы много работаете со стандартной библиотекой и используете заголовочный файл `<algorithm>`, вы определенно пишете много функциональных объектов. Начиная с C++11 вы можете использовать лямбды с алгоритами стандартной библиотеки. В более ранних версиях стандарта C++ вы можете создавать функциональные объекты, используя такие адаптеры, как `bind1st`, `bind2nd`, `ptr_fun`, `mem_fun`, `mem_fun_ref`, или можете писать функциональные объекты вручную (потому что адаптеры выглядят пугающе). Хорошая новость: библиотеку `Boost.Bind` можно использовать вместо уродливых адаптеров, и она обеспечивает более понятный синтаксис.

Подготовка

Знание стандартных библиотечных функций и алгоритмов будет полезно.

Как это делается...

Давайте посмотрим на примеры использования библиотеки `Boost.Bind` и аналогичные примеры с лямбдами C++11:

1. Все примеры в этом рецепте требуют подключения следующих заголовочных файлов:

```
// Содержит функцию boost::bind и _1, _2, _3....
#include <boost/bind.hpp>

// Полезные вещи, которые нужны образцам.
#include <boost/array.hpp>
#include <algorithm>
#include <functional>
#include <string>
#include <cassert>
```

2. Подсчитаем значения больше 5, как показано в приведенном ниже коде:

```
void sample1() {
    const boost::array<int, 12> v = {{
        1, 2, 3, 4, 5, 6, 7, 100, 99, 98, 97, 96
    }};

    const std::size_t count0 = std::count_if(v.begin(), v.end(),
        [](int x) { return 5 < x; }
    );
    const std::size_t count1 = std::count_if(v.begin(), v.end(),
        boost::bind(std::less<int>(), 5, _1)
    );
    assert(count0 == count1);
}
```

3. Вот как можно сосчитать пустые строки:

```
void sample2() {
    const boost::array<std::string, 3> v = {{
        "We ", "are", " the champions!"
    }};
```

```
    const std::size_t count0 = std::count_if(v.begin(), v.end(),
        [](const std::string& s) { return s.empty(); }
    );
    const std::size_t count1 = std::count_if(v.begin(), v.end(),
        boost::bind(&std::string::empty, _1)
    );
    assert(count0 == count1);
}
```

4. Теперь давайте посчитаем строки длиной менее 5:

```
void sample3() {
    const boost::array<std::string, 3> v = {{
        "We ", "are", " the champions!"
    }};

    const std::size_t count0 = std::count_if(v.begin(), v.end(),
        [](const std::string& s) { return s.size() < 5; }
    );
    const std::size_t count1 = std::count_if(v.begin(), v.end(),
        boost::bind(
            std::less<std::size_t>(),
            boost::bind(&std::string::size, _1),
            5
        )
    );
    assert(count0 == count1);
}
```

5. Сравним строки:

```
void sample4() {
    const boost::array<std::string, 3> v = {{
        "We ", "are", " the champions!"
    }};
    std::string s(
        "Expensive copy constructor is called when binding"

    );
    const std::size_t count0 = std::count_if(v.begin(), v.end(),
        [&s](const std::string& x) { return x < s; }
    );
    const std::size_t count1 = std::count_if(v.begin(), v.end(),
        boost::bind(std::less<std::string>(), _1, s)
    );
    assert(count0 == count1);
}
```

Как это работает...

Функция `boost::bind` возвращает функциональный объект, в котором хранятся копии всех аргументов функции. Когда выполняется фактический вызов функции `operator()`, сохраненные параметры передаются в исходный функциональный объект вместе с параметрами, переданными во время вызова.

Дополнительно...

Взгляните на предыдущие примеры. Когда мы вызываем `boost::bind`, то копируем значения в функциональный объект. Для некоторых классов это затратная операция. Есть ли способ обойти копирование?

Да, есть! Здесь нам поможет библиотека `Boost.Ref`! Она содержит две функции, `boost::ref()` и `boost::cref()`, первая из которых позволяет сохранять параметр в качестве ссылки, а вторая сохраняет параметр в качестве константной ссылки. Функции `ref()` и `cref()` просто создают объект типа `reference_wrapper<T>` или `reference_wrapper<const T>`, который умеет неявно преобразовываться в ссылочный тип. Давайте внесем изменения в наши последние примеры:

```
#include <boost/ref.hpp>

void sample5() {
    const boost::array<std::string, 3> v = {{
        "We ", "are", " the champions!"
    }};
    std::string s(
        "Expensive copy constructor is NOT called when binding"
    );

    const std::size_t count1 = std::count_if(v.begin(), v.end(),
        boost::bind(std::less<std::string>(), _1, boost::cref(s))
    );
    // ...
}
```

Вы также можете переупорядочивать, игнорировать и дублировать параметры функции, используя функцию `bind`:

```
void sample6() {
    const auto twice = boost::bind(std::plus<int>(), _1, _1);
    assert(twice(2) == 4);

    const auto minus_from_second = boost::bind(std::minus<int>(), _2, _1);
    assert(minus_from_second(2, 4) == 2);

    const auto sum_second_and_third = boost::bind(
        std::plus<int>(), _2, _3
    );
    assert(sum_second_and_third(10, 20, 30) == 50);
}
```

Функции `ref`, `cref` и `bind` приняты в стандарт C++11 и определены в заголовке `<functional>` в пространстве имен `std::`. Все эти функции не выделяют память динамически и не используют виртуальных функций. Возвращаемые ими объекты легко оптимизируются толковыми компиляторами.

Реализации этих функций в стандартной библиотеке могут иметь дополнительные оптимизации для сокращения времени компиляции. Вы можете использовать версии функций `bind`, `ref`, `cref` стандартных библиотек с любой библиотекой Boost или даже смешивать версии из Boost и стандартных библиотек.

Если вы применяете компилятор C++14, то используйте универсальные лямбда-выражения вместо `std::bind` и `boost::bind`, поскольку их проще понять. В отличие от `boost::bind`, лямбды с C++17 можно использовать со спецификатором типа constexpr.

См. также

В официальной документации содержится еще больше примеров и описание расширенных функций: http://boost.org/libs/bind.

Получение удобочитаемого имени типа

Часто возникает необходимость получить читабельное имя типа во время выполнения:

```
#include <iostream>
#include <typeinfo>

template <class T>
void do_something(const T& x) {
    if (x == 0) {
        std::cout << "Error: x == 0. T is " << typeid(T).name()
        << std::endl;
    }
    // ...
}
```

Тем не менее приведенный ранее пример не является особо переносимым. Он не работает, если динамическая идентификация типа данных (RTTI) отключена, и не всегда выдает красивое удобочитаемое имя. На некоторых платформах такой код будет давать на выходе только `i` или `d`.

Ситуация ухудшается, если нам нужна полная спецификация типа без удаления квалификаторов `const`, `volatile` и ссылок:

```
void sample1() {
    auto&& x = 42;
    std::cout << "x is "
              << typeid(decltype(x)).name()
              << std::endl;
}
```

К сожалению, предыдущий код в лучшем случае выведет `int`, а это не то, что мы ожидали увидеть.

Подготовка

Для этого рецепта требуется базовое знание C++.

Как это делается...

В первом примере нам нужно удобочитаемое имя типа без квалификаторов. Нам поможет библиотека `Boost.TypeIndex`:

```
#include <iostream>
#include <boost/type_index.hpp>

template <class T>
void do_something_again(const T& x) {
    if (x == 0) {
        std::cout << "x == 0. T is " << boost::typeindex::type_id<T>()
                  << std::endl;
    }
    // ...
}
```

Во втором примере нам нужно оставить квалификаторы, поэтому нужно вызвать немного другую функцию из той же библиотеки:

```
#include <boost/type_index.hpp>

void sample2() {
    auto&& x = 42;
    std::cout << "x is "
              << boost::typeindex::type_id_with_cvr<decltype(x)>()
              << std::endl;
}
```

Как это работает...

Библиотека Boost.TypeIndex знает о внутреннем устройстве разных компиляторов и обладает знаниями о наиболее эффективном способе создания удобочитаемого имени для типа. Если вы предоставляете тип в качестве параметра шаблона, библиотека гарантирует, что все возможные вычисления, связанные с типами, будут выполняться во время компиляции, и код будет работать, даже если динамическая идентификация типа данных (RTTI) отключена.

cvr в boost::typeindex::type_id_with_cvr расшифровывается как const, volatile и reference. Эта функция гарантирует, что информация о квалификаторах и ссылках окажется в результате функции.

Дополнительно...

Все функции boost::typeindex::type_id* возвращают экземпляры boost::typeindex::type_index. Этот тип очень похож на std::type_index; однако у него есть метод raw_name() для получения необработанного имени типа и метод pretty_name() для получения человекочитаемого имени типа.

Даже в самых новых стандартах C++ std::type_index и std::type_info возвращают специфичные для платформы представления имен типов, которые довольно сложно декодировать или использовать переносимо.

В отличие от метода стандартной библиотеки typeid(), некоторые классы из библиотеки Boost.TypeIndex можно использовать со спецификатором типа constexpr. Это означает, что вы можете получить текстовое представление вашего типа во время компиляции, если используете специфический класс boost::typeindex::ctti_type_index.

Пользователи могут создавать собственные реализации динамической идентификации типов данных, используя библиотеку Boost.TypeIndex. Это мо-

жет быть полезно для разработчиков встраиваемых систем и для приложений, которым требуется чрезвычайно эффективная динамическая идентификация типа данных, настроенная на определенные типы.

См. также

Документация по расширенным функциям и дополнительные примеры доступны на странице http://www.boost.org/libs/type_index.

Использование эмуляции перемещения C++11

Одна из главных особенностей стандарта C++11 – это rvalue-ссылки, та же move-семантика. Эта особенность позволяет нам изменять временные объекты, «крадя» у них ресурсы. Как вы можете догадаться, стандарт C++03 не имеет rvalue-ссылок, но, используя библиотеку `Boost.Move`, можно написать переносимый код, который эмулирует их.

Подготовка

Настоятельно рекомендуется, чтобы вы хотя бы были знакомы с основами rvalue-ссылок C++11.

Как это делается...

1. Представьте, что у вас есть класс с несколькими полями, некоторые из которых являются контейнерами стандартной библиотеки:

```
namespace other {
    class characteristics{};
}
struct person_info {
    std::string name_;
    std::string second_name_;
    other::characteristics characteristic_;
    // ...
};
```

2. Настало время добавить перемещающее присвоение (move assignment) и перемещающий конструктор (move constructor)! Помните, что в C++03 стандартные контейнеры не имеют перемещающего конструктора и перемещающего присваивания.
3. Правильная реализация перемещающего присваивания аналогична перемещающему конструированию временной переменной от входного аргумента и обмену значениями с `this`. Правильная реализация конструктора перемещения близка к конструированию объекта по умолчанию и обмену значениями с входным параметром. Итак, начнем с функции-члена для обмена значениями `swap`:

```
#include <boost/swap.hpp>

void person_info::swap(person_info& rhs) {
    name_.swap(rhs.name_);
```

```
    second_name_.swap(rhs.second_name_);
    boost::swap(characteristic_, rhs.characteristic_);
}
```

4. Теперь поместите следующий макрос в раздел `private`:

```
BOOST_COPYABLE_AND_MOVABLE(person_info)
```

5. Напишите копирующий конструктор.
6. Напишите копирующее присваивание, принимая параметр как

```
BOOST_COPY_ASSIGN_REF(person_info)
```

7. Напишите конструктор перемещения и присваивание перемещением, принимая параметр как `BOOST_RV_REF (person_info)`:

```
struct person_info {
    // Поля объявляются здесь;
    // ...
private:
    BOOST_COPYABLE_AND_MOVABLE(person_info)
public:
    // Чтобы было проще, мы предположим, что конструктор по умолчанию
    // person_info и функцию swap очень быстро и дешево вызывать;
    person_info();

    person_info(const person_info& p)
        : name_(p.name_)
        , second_name_(p.second_name_)
        , characteristic_(p.characteristic_)
    {}

    person_info(BOOST_RV_REF(person_info) person) {
        swap(person);
    }

    person_info& operator=(BOOST_COPY_ASSIGN_REF(person_info) person) {
        person_info tmp(person);
        swap(tmp);
        return *this;
    }

    person_info& operator=(BOOST_RV_REF(person_info) person) {
        person_info tmp(boost::move(person));
        swap(tmp);
        return *this;
    }

    void swap(person_info& rhs);
};
```

8. Теперь у нас есть переносимая быстрая реализация конструктора и оператора перемещения для класса `person_info`.

Как это работает...

Вот пример того, как можно воспользоваться move-семантикой:

```
int main() {
    person_info vasya;
    vasya.name_ = "Vasya";
    vasya.second_name_ = "Snow";

    person_info new_vasya(boost::move(vasya));
    assert(new_vasya.name_ == "Vasya");
    assert(new_vasya.second_name_ == "Snow");
    assert(vasya.name_.empty());
    assert(vasya.second_name_.empty());

    vasya = boost::move(new_vasya);
    assert(vasya.name_ == "Vasya");
    assert(vasya.second_name_ == "Snow");
    assert(new_vasya.name_.empty());
    assert(new_vasya.second_name_.empty());
}
```

Библиотека `Boost.Move` реализована очень эффективно. Когда используется компилятор C++11, все макросы для эмуляции rvalues разворачиваются в обычные rvalue C++11, в противном случае (на компиляторах C++03) rvalues эмулируются.

Дополнительно...

Вы обратили внимание на вызов функции `boost::swap`? Это действительно полезная служебная функция, которая сначала ищет функцию `swap` в пространстве имен переменной (в нашем примере это пространство имен `other::`), и если соответствующей функции нет, она использует `std::swap`.

См. также

- Дополнительную информацию о реализации эмуляции можно найти на сайте Boost и в исходниках библиотеки `Boost.Move` по адресу http://boost.org/libs/move;
- библиотека `Boost.Utility` содержит функцию `boost::swap`, и у нее имеется множество полезных функций и классов. На странице http://boost.org/libs/utility вы найдете документацию по ней;
- рецепт «*Инициализация базового класса членом класса-наследника*» главы 2 «Управление ресурсами»;
- рецепт «*Создание некопируемого класса*»;
- в рецепте «*Создание некопируемого, но перемещаемого класса*» есть больше информации о библиотеке `Boost.Move` и приводятся примеры того, как можно использовать перемещаемые объекты в контейнерах переносимым и эффективным способом.

Создание некопируемого класса

Вы почти наверняка сталкивались с определенными ситуациями, когда классу принадлежат некие ресурсы, которые не следует копировать по техническим причинам:

```
class descriptor_owner {
    void* descriptor_;

public:
    explicit descriptor_owner(const char* params);

    ~descriptor_owner() {
        system_api_free_descriptor(descriptor_);
    }
};
```

Компилятор C++ для предыдущего примера сгенерирует копирующий конструктор и оператор присваивания, поэтому потенциальный пользователь класса `descriptor_owner` сможет написать и скомпилировать вот такие ужасные вещи:

```
void i_am_bad() {
    descriptor_owner d1("O_o");
    descriptor_owner d2("^_^");

    // Дескриптор d2 не был правильно освобожден
    d2 = d1;

    // деструктор d2 освободит дескриптор
    // деструктор d1 попытается освободить уже освобожденный дескриптор
}
```

Подготовка

Все, что требуется для этого рецепта, – базовые знания C++.

Как это делается...

Чтобы избежать таких ситуаций, был придуман класс `boost::noncopyable`. Если вы наследуете от него свой собственный класс, копирующий конструктор и оператор присваивания не будут сгенерированы компилятором C++:

```
#include <boost/noncopyable.hpp>
class descriptor_owner_fixed : private boost::noncopyable {
    // ...
```

Теперь пользователь не сможет делать плохие вещи:

```
void i_am_good() {
    descriptor_owner_fixed d1("O_o");
    descriptor_owner_fixed d2("^_^");

    // Компиляции не будет
    d2 = d1;
```

```
    // И здесь компиляции не будет
    descriptor_owner_fixed d3(d1);
}
```

Как это работает...

Искушенный читатель заметит, что можно достичь точно такого же результата:

- написав копирующий конструктор и оператор присваивания для `descriptor_owning_fixed`;
- определив их без фактической реализации;
- явно удалив их, используя синтаксис C++11 `= delete`.

Да, вы правы. В зависимости от возможностей вашего компилятора класс `boost::noncopyable` выбирает самый подходящий способ сделать класс некопируемым.

`boost::noncopyable` также служит хорошей документацией для вашего класса. С ним никогда не возникает таких вопросов, как «Определено ли тело копирующего конструктора где-то еще?» или «Есть ли у него нестандартный копирующий конструктор (с неконстантным ссылочным параметром)?».

См. также

- Рецепт «*Создание некопируемого, но перемещаемого класса*» даст вам идеи о том, как разрешить уникальное владение ресурсом в C++03 путем его перемещения;
- вы можете найти много полезных функций и классов в официальной документации библиотеки `Boost.Core` по адресу http://boost.org/libs/core;
- рецепт «*Инициализация базового класса членом класса-наследника*» главы 2 «Управление ресурсами»;
- рецепт «*Использование эмуляции перемещения в C++11*».

Создание некопируемого, но перемещаемого класса

Теперь представьте следующую ситуацию: у нас есть ресурс, который нельзя скопировать, который должен быть правильно освобожден в деструкторе, и нужно вернуть его из функции:

```
descriptor_owner construct_descriptor()
{
     return descriptor_owner("Construct using this string");
}
```

На самом деле такие ситуации можно обойти, используя метод `swap`:

```
void construct_descriptor1(descriptor_owner& ret)
{
    descriptor_owner("Construct using this string").swap(ret);
}
```

Однако такой обходной путь не позволяет нам использовать класс `descriptor_owner` в контейнерах. Да и выглядит подобное решение ужасно!

Подготовка

Настоятельно рекомендуется, чтобы вы были по крайней мере знакомы с основами rvalue-ссылок C++11. Также рекомендуется прочитать рецепт «*Использование эмуляции перемещения в C++11*».

Как это делается...

Те читатели, которые используют стандарт C++11, уже знают о классах, которые можно перемещать, но не копировать (например, `std::unique_ptr` или `std::thread`). Используя такой подход, мы можем создать класс `descriptor_owner`, который можно перемещать, но не копировать:

```
class descriptor_owner1 {
    void* descriptor_;

public:
    descriptor_owner1()
      : descriptor_(nullptr)
    {}

    explicit descriptor_owner1(const char* param);

    descriptor_owner1(descriptor_owner1&& param)
        : descriptor_(param.descriptor_)
    {
        param.descriptor_ = nullptr;
    }

    descriptor_owner1& operator=(descriptor_owner1&& param) {
        descriptor_owner1 tmp(std::move(param));
        std::swap(descriptor_, tmp.descriptor_);
        return *this;
    }

    void clear() {
        free(descriptor_);
        descriptor_ = nullptr;
    }

    bool empty() const {
        return !descriptor_;
    }

    ~descriptor_owner1() {
        clear();
    }
};

// GCC компилирует это в C++11 и более поздних режимах.
descriptor_owner1 construct_descriptor2() {
    return descriptor_owner1("Construct using this string");
}
```

```
void foo_rv() {
    std::cout << "C++11\n";
    descriptor_owner1 desc;
    desc = construct_descriptor2();
    assert(!desc.empty());
}
```

Это будет работать только на компиляторах, совместимых с C++11. Подходящий момент для Boost.Move! Давайте внесем изменения в наш пример, чтобы его можно было использовать на компиляторах C++03.

Чтобы написать перемещаемый, но некопируемый тип в переносимом синтаксисе, нужно выполнить следующие простые шаги:

1. Поместить макрос BOOST_MOVABLE_BUT_NOT_COPYABLE(имя класса) в раздел private:

```
#include <boost/move/move.hpp>

class descriptor_owner_movable {
    void* descriptor_;

    BOOST_MOVABLE_BUT_NOT_COPYABLE(descriptor_owner_movable)
```

2. Написать перемещающий конструктор и перемещающее присваивание, принимая параметр как BOOST_RV_REF(имя класса):

```
public:
    descriptor_owner_movable()
        : descriptor_(NULL)
    {}

    explicit descriptor_owner_movable(const char* param)
        : descriptor_(strdup(param))
    {}

    descriptor_owner_movable(
        BOOST_RV_REF(descriptor_owner_movable) param
    ) BOOST_NOEXCEPT
        : descriptor_(param.descriptor_)
    {
        param.descriptor_ = NULL;
    }

    descriptor_owner_movable& operator=(
        BOOST_RV_REF(descriptor_owner_movable) param) BOOST_NOEXCEPT
    {
        descriptor_owner_movable tmp(boost::move(param));
        std::swap(descriptor_, tmp.descriptor_);
        return *this;
    }

    // ...
};
```

```
    descriptor_owner_movable construct_descriptor3() {
        return descriptor_owner_movable("Construct using this string");
    }
```

Как это работает...

Теперь у нас есть перемещаемый, но некопируемый класс, который можно использовать даже в компиляторах C++03 и с контейнерами из `Boost.Containers`:

```
#include <boost/container/vector.hpp>
#include <your_project/descriptor_owner_movable.h>

int main() {
    // Приведенный ниже код будет работать на компиляторах C++11 и C++03
    descriptor_owner_movable movable;
    movable = construct_descriptor3();
    boost::container::vector<descriptor_owner_movable> vec;
    vec.resize(10);
    vec.push_back(construct_descriptor3());

    vec.back() = boost::move(vec.front());
}
```

К сожалению, контейнеры стандартной библиотеки C++03 по-прежнему не смогут его использовать (вот почему в предыдущем примере мы использовали вектор из `Boost.Containers`).

Дополнительно...

Если вы хотите использовать `Boost.Containers` на компиляторах C++03, а контейнеры стандартной библиотеки – на компиляторах C++11, можете выполнить следующий простой трюк. Добавьте в свой проект файл заголовка следующего содержания:

```
// your_project/vector.hpp
// Указание лицензии и авторских прав

// include guards
#ifndef YOUR_PROJECT_VECTOR_HPP
#define YOUR_PROJECT_VECTOR_HPP

// Содержит макрос BOOST_NO_CXX11_RVALUE_REFERENCES
#include <boost/config.hpp>

#if !defined(BOOST_NO_CXX11_RVALUE_REFERENCES)
// rvalue-ссылки доступны
#include <vector>

namespace your_project_namespace {
  using std::vector;
} // your_project_namespace

#else
// rvalue-ссылки недоступны
#include <boost/container/vector.hpp>
```

```
namespace your_project_namespace {
  using boost::container::vector;
} // your_project_namespace

#endif // !defined(BOOST_NO_CXX11_RVALUE_REFERENCES)
#endif // YOUR_PROJECT_VECTOR_HPP
```

Теперь вы можете включить в код `<your_project/vector.hpp>` и использовать вектор из пространства имен `your_project_namespace`:

```
int main() {
    your_project_namespace::vector<descriptor_owner_movable> v;
    v.resize(10);
    v.push_back(construct_descriptor3());
    v.back() = boost::move(v.front());
}
```

См. также

- В рецепте «*Уменьшение размера кода и повышение производительности пользовательского типа в C++11*» главы 10 «Сбор информации о платформе и компиляторе» содержится дополнительная информация о `noexcept` и `BOOST_NOEXCEPT`;
- более подробную информацию о библиотеке `Boost.Move` можно найти на сайте Boost: http://www.boost.org/libs/move.

Использование алгоритмов C++14 и C++11

В C++11 есть несколько новых классных алгоритмов в заголовочном файле `<algorithm>`. В C++14 и C++17 их стало еще больше. Если вы застряли на компиляторе для стандарта, предшествующего C++11, то вам необходимо писать эти алгоритмы с нуля. Например, если вы хотите вывести ASCII-символы от 65 до 125, то должны написать следующий код:

```
#include <boost/array.hpp>

boost::array<unsigned char, 60> chars_65_125_pre11() {
    boost::array<unsigned char, 60> res;

    const unsigned char offset = 65;
    for (std::size_t i = 0; i < res.size(); ++i) {
        res[i] = i + offset;
    }

    return res;
}
```

Подготовка

Для этого рецепта требуются базовые знания C++, а также базовые знания библиотеки `Boost.Array`.

Как это делается...

Библиотека Boost.Algorithm содержит все новые алгоритмы C++11 и C++14. Используя ее, можно переписать предыдущий пример следующим образом:

```
#include <boost/algorithm/cxx11/iota.hpp>
#include <boost/array.hpp>

boost::array<unsigned char, 60> chars_65_125() {
    boost::array<unsigned char, 60> res;
    boost::algorithm::iota(res.begin(), res.end(), 65);
    return res;
}
```

Как это работает...

Как вы, наверное, знаете, в библиотеке Boost.Algorithm есть отдельный заголовочный файл для каждого алгоритма. Просто откройте нужный заголовочный файл и используйте необходимую функцию.

Дополнительно...

Скучно, когда есть библиотека, которая просто реализует алгоритмы из стандарта C++. Это не новаторство; это не в традициях Boost! Вот почему вы можете найти в Boost.Algorithm функции, которые не являются частью стандарта C++. Вот, например, функция, которая преобразует данные в шестнадцатеричное представление:

```
#include <boost/algorithm/hex.hpp>
#include <iterator>
#include <iostream>

void to_hex_test1() {
    const std::string data = "Hello word";
    boost::algorithm::hex(
        data.begin(), data.end(),
        std::ostream_iterator<char>(std::cout)
    );
}
```

Предыдущий код выводит следующее:

```
48656C6C6F20776F7264
```

Что еще интереснее, все функции в Boost.Algorithm имеют дополнительные перегрузки, которые вместо двух итераторов принимают первым параметром диапазон. **Диапазон** – это концепция от **Ranges TS**. Массивы и контейнеры с функциями .begin() и .end() соответствуют концепции диапазона. Зная это, предыдущий пример можно сократить:

```
#include <boost/algorithm/hex.hpp>
#include <iterator>
#include <iostream>
```

```
void to_hex_test2() {
    const std::string data = "Hello word";
    boost::algorithm::hex(
        data,
        std::ostream_iterator<char>(std::cout)
    );
}
```

C++17-библиотека Boost.Algorithm скоро будет расширена новыми алгоритмами и функциями C++20, такими как алгоритмы, которые можно использовать в constexpr-функциях. Следите за этой библиотекой, поскольку однажды она может стать готовым решением проблемы, с которой вы имеете дело.

См. также

- В официальной документации для библиотеки Boost.Algorithm содержится полный список функций и краткие описания для них: http://boost.org/libs/algorithm;
- экспериментируйте с новыми алгоритмами в режиме онлайн: http://apolukhin.github.io/Boost-Cookbook.

Глава 2

Управление ресурсами

Темы, которые мы рассмотрим в этой главе:

- управление указателями на классы, которые не покидают область видимости;
- подсчет указателей на классы;
- управление указателями на массивы, которые не покидают область видимости;
- подсчет указателей на массивы;
- хранение любых функциональных объектов в переменной;
- передача указателя на функцию;
- хранение любых лямбда-функций C++11 в переменной;
- контейнеры указателей;
- инициализация базового класса членом класса-наследника.

Вступление

В этой главе мы продолжим работу с типами данных, представленными библиотеками Boost, в основном сосредоточившись на работе с указателями. Мы увидим, как с легкостью управлять ресурсами и использовать тип данных, способный хранить любые функциональные объекты, функции и лямбда-выражения. После прочтения этой главы ваш код станет более надежным, а утечки памяти уйдут в прошлое.

Управление указателями на классы, которые не покидают область видимости

Иногда нам необходимо динамически выделить память и создать класс в этой памяти. Вот где начинаются проблемы. Посмотрите на приведенный ниже код:

```
bool foo1() {
    foo_class* p = new foo_class("Some data");

    const bool something_else_happened = some_function1(*p);
    if (something_else_happened) {
        delete p;
        return false;
  }
  some_function2(p);
```

```
    delete p;
    return true;
}
```

На первый взгляд все выглядит правильно. Но что, если `some_function1()` или `some_function2()` выбросит исключение? В этом случае `p` не будет удален. Давайте исправим это следующим образом:

```
bool foo2() {
    foo_class* p = new foo_class("Some data");
    try {
        const bool something_else_happened = some_function1(*p);
        if (something_else_happened) {
            delete p;
            return false;
        }
        some_function2(p);
    } catch (...) {
        delete p;
        throw;
    }
    delete p;
    return true;
}
```

Теперь код правильный, но выглядит уродливо, и его трудно читать. Можно ли сделать это как-нибудь получше?

Подготовка

Требуются базовые знания C++ и поведения кода во время исключений.

Как это делается...

Просто взгляните на библиотеку `Boost.SmartPtr`. Существует класс `boost::scoped_ptr`, который может вам помочь:

```
#include <boost/scoped_ptr.hpp>

bool foo3() {
    const boost::scoped_ptr<foo_class> p(new foo_class("Some data"));

    const bool something_else_happened = some_function1(*p);
    if (something_else_happened) {
        return false;
    }
    some_function2(p.get());
    return true;
}
```

Теперь нет никаких шансов, что произойдет утечка ресурса, а исходный код станет намного понятнее.

Если у вас есть контроль над some_function2 foo_class *), вы можете переписать ее, чтобы принять ссылку на foo_class вместо указателя. Интерфейс со ссылками более интуитивен, чем интерфейс с указателями (если в вашей компании нет специального соглашения, согласно которому выходные параметры принимаются только по указателю).

Кстати, в библиотеке Boost.Move также есть класс boost::movelib::unique_ptr, который можно использовать вместо класса boost::scoped_ptr:

```
#include <boost/move/make_unique.hpp>

bool foo3_1() {
    const boost::movelib::unique_ptr<foo_class> p
        = boost::movelib::make_unique<foo_class>("Some data");
    const bool something_else_happened = some_function1(*p);
    if (something_else_happened) {
        return false;
    }
    some_function2(p.get());
    return true;
}
```

Как это работает...

boost::scoped_ptr<T> и boost::movelib::unique_ptr являются типичными представителями программной идиомы **RAII** (Получение ресурса есть инициализация – англ. **Resource Acquisition Is Initialization**). Когда выбрасывается исключение или переменная выходит за пределы области видимости, стек сворачивается, и вызывается деструктор. В деструкторе классы scoped_ptr<T> и unique_ptr<T> вызывают delete для указателя, который они хранят. Поскольку оба этих класса по умолчанию вызывают delete, можно безопасно хранить класс-наследник по указателю на базовый класс, если деструктор базового класса является виртуальным:

```
#include <iostream>
#include <string>

struct base {
    virtual ~base(){}
};

class derived: public base {
    std::string str_;

public:
    explicit derived(const char* str)
        : str_(str)
    {}
    ~derived() /*override*/ {
        std::cout << "str == " << str_ << '\n';
    }
};
```

```
void base_and_derived() {
    const boost::movelib::unique_ptr<base> p1(
        boost::movelib::make_unique<derived>("unique_ptr")
    );

    const boost::scoped_ptr<base> p2(
        new derived("scoped_ptr")
    );
}
```

Запуск функции base_and_derived() приведет к следующему выводу:

```
str == scoped_ptr
str == unique_ptr
```

В C++ деструкторы для объектов вызываются в обратном порядке конструирования. Вот почему деструктор scoped_ptr был вызван перед деструктором unique_ptr.

Объекты класса boost::scoped_ptr<T> нельзя ни копировать, ни перемещать. boost::movelib::unique_ptr является классом, который можно перемещать, но не копировать. Он использует эмуляцию move-семантики в старых компиляторах, не поддерживающих C++11. Оба класса хранят лишь указатель на ресурс, которым они владеют, и не требуют, чтобы T был полным типом (T может быть объявлен предварительно – типом forward declared).

Некоторые компиляторы не предупреждают об удалении неполного типа, что может привести к ошибкам, которые трудно обнаружить. К счастью, это не относится к классам Boost, которые имеют проверки времени компиляции для таких случаев. Это делает scoped_ptr и unique_ptr идеальными классами для реализации идиомы **Pimpl** (Указатель на реализацию – англ. Pointer to implementation):

```
// В файле заголовка:
struct public_interface {
    // ...
private:
    struct impl; // Forward declaration.
    boost::movelib::unique_ptr<impl> impl_;
};
```

Дополнительно...

Эти классы очень быстрые. Компилятор хорошо оптимизирует код, который использует scoped_ptr и unique_ptr. Вы не сможете получить большую производительность, используя ручное управление памятью.

В C++11 есть класс std::unique_ptr<T, D>, который уникально владеет ресурсом и ведет себя точно так же, как boost::movelib::unique_ptr<T, D>.

В стандартной библиотеке C++ нет boost::scoped_ptr<T>, но вместо этого можно использовать const std::unique_ptr<T>. Единственная разница состоит в том, что boost::scoped_ptr<T> по-прежнему может вызывать метод reset(), в отличие от const std::unique_ptr<T>.

См. также

- Документация по библиотеке Boost.SmartPtr содержит множество примеров и другую полезную информацию обо всех классах умных указателей. О них можно прочитать на странице http://boost.org/libs/smart_ptr;
- документация и примеры библиотеки Boost.Move могут помочь вам, если вы используете эмуляцию перемещения с классом boost::movelib::unique_ptr: http://boost.org/libs/move.

Подсчет указателей на классы

Представьте, что у вас есть некая динамически выделенная структура, содержащая данные, и вы хотите обрабатывать ее в разных потоках выполнения. Код для этого выглядит следующим образом:

```
#include <boost/thread.hpp>
#include <boost/bind.hpp>

void process1(const foo_class* p);
void process2(const foo_class* p);
void process3(const foo_class* p);

void foo1() {
    while (foo_class* p = get_data()) // C way
    {
        // Скоро здесь будет слишком много потоков, см. рецепт «Параллельное выполнение
        // различных задач», где приводится хороший способ избежать
        // неконтролируемого роста потоков.
        boost::thread(boost::bind(&process1, p))
            .detach();
        boost::thread(boost::bind(&process2, p))
            .detach();
        boost::thread(boost::bind(&process3, p))
            .detach();

        // delete p; Oops!!!!
    }
}
```

Мы не можем освободить p в конце цикла while, потому что ресурсы этой переменной все еще могут использоваться потоками, выполняющими функции process. Эти функции не могут удалить p, потому что они не знают, перестали ли другие потоки использовать эту переменную.

Подготовка

В этом рецепте используется библиотека Boost.Thread, которая не является библиотекой header-only. Ваша программа должна линковаться с библиотеками boost_thread, boost_chrono и boost_system. Убедитесь, что вы понимаете концепцию потоков, прежде чем читать дальше. Обратитесь к разделу «См. также», где приводятся ссылки на рецепты, которые описывают потоки.

Вам также понадобятся базовые знания boost::bind или std::bind, которые почти одинаковы.

Как это делается...

Как вы уже догадались, в Boost (и C++11) есть класс, который может помочь вам справиться с этой проблемой. Он называется `boost::shared_ptr`. Его можно использовать следующим образом:

```
#include <boost/shared_ptr.hpp>

void process_sp1(const boost::shared_ptr<foo_class>& p);
void process_sp2(const boost::shared_ptr<foo_class>& p);
void process_sp3(const boost::shared_ptr<foo_class>& p);

void foo2() {
    typedef boost::shared_ptr<foo_class> ptr_t;
    ptr_t p;
    while (p = ptr_t(get_data())) // Как в C
    {
        boost::thread(boost::bind(&process_sp1, p))
            .detach();
        boost::thread(boost::bind(&process_sp2, p))
            .detach();
        boost::thread(boost::bind(&process_sp3, p))
            .detach();
        // Не нужно ничего делать
    }
}
```

Еще один пример:

```
#include <string>
#include <boost/smart_ptr/make_shared.hpp>

void process_str1(boost::shared_ptr<std::string> p);
void process_str2(const boost::shared_ptr<std::string>& p);
void foo3() {
    boost::shared_ptr<std::string> ps = boost::make_shared<std::string>(
        "Guess why make_shared<std::string> "
        "is faster than shared_ptr<std::string> "
        "ps(new std::string('this string'))"
    );

   boost::thread(boost::bind(&process_str1, ps))
           .detach();
   boost::thread(boost::bind(&process_str2, ps))
           .detach();
}
```

Как это работает...

Внутри класса `shared_ptr` есть атомарный счетчик ссылок. Когда вы копируете его, счетчик увеличивается, а когда вызывается его деструктор, счетчик уменьшается. Когда количество ссылок равно нулю, вызывается `delete` для объекта, на который указывает `shared_ptr`.

Теперь давайте выясним, что происходит в случае boost::thread(boost::bind(&process_sp1, p)). Функция process_sp1 принимает параметр в качестве ссылки. Так почему он не освобождается, когда мы выходим из цикла while? Ответ прост. Функциональный объект, возвращаемый функцией bind(), содержит копию shared_ptr, а это означает, что данные, на которые указывает p, не будут освобождены, пока функциональный объект не будет уничтожен. Функциональный объект копируется в поток и живет до тех пор, пока не завершится выполнение потока.

Вернемся к shared_ptr<std::string> ps(new int(0)). В этом случае у нас есть два вызова new:

- при создании указателя на целое число с помощью new int(0);
- при создании внутреннего счетчика ссылок класса shared_ptr, который выделяется в куче.

Используйте make_shared<T>, чтобы у вас был только один вызов new. make_shared<T> выделяет один фрагмент памяти и создает в нем и атомарный счетчик, и объект T.

Дополнительно...

Счетчик атомарных ссылок гарантирует правильное поведение класса shared_ptr в потоках, но вы должны помнить, что атомарные операции выполняются не так быстро, как неатомарные. Класс shared_ptr меняет атомарную переменную при присваиваниях, вызовах копирующего конструктора и уничтожении ненулевого shared_ptr. Это означает, что в компиляторах, совместимых с C++11, вы можете уменьшить количество атомарных операций, по возможности применяя перемещающее конструирование и перемещающие присваивания. Просто используйте shared_ptr<T> p1(std::move(p)), если вы больше не будете применять переменную p.

Если вы не собираетесь изменять указываемое значение, рекомендуется сделать его константным. Просто добавьте слово const к шаблонному параметру умного указателя, и компилятор позаботится о том, чтобы вы не модифицировали память:

```
void process_cstr1(boost::shared_ptr<const std::string> p);
void process_cstr2(const boost::shared_ptr<const std::string>& p);
void foo3_const() {
    boost::shared_ptr<const std::string> ps
        = boost::make_shared<const std::string>(
            "Some immutable string"
        );

    boost::thread(boost::bind(&process_cstr1, ps))
        .detach();
    boost::thread(boost::bind(&process_cstr2, ps))
        .detach();
    // *ps = "qwe"; // Ошибка компиляции, string константен!
}
```

Вас смущает слово const? Вот сопоставление константности умного указателя с константностью простого указателя:

shared_ptr<T>	T*
shared_ptr<const T>	const T*
const shared_ptr<T>	T* const
const shared_ptr<const T>	const T* const

Класс shared_ptr и функция make_shared являются частью C++11, и они объявлены в заголовочном файле <memory> в пространстве имен std::. Они имеют почти те же характеристики, что и Boost-версии.

См. также

- Обратитесь к главе 5 «*Многопоточность*» для получения дополнительной информации о Boost.Thread и атомарных операциях;
- для получения дополнительной информации о Boost.Bind *см.* рецепт «*Привязка и переупорядочение параметров функции*» главы 1 «Приступаем к написанию приложения»;
- обратитесь к рецепту «*Приведение умных указателей*» главы 3 для получения информации о том, как преобразовать shared_ptr<U> в shared_ptr<T>;
- документация по библиотеке Boost.SmartPtr содержит множество примеров и другую полезную информацию обо всех классах умного указателя. Перейдите по ссылке http://boost.org/libs/smart_ptr, чтобы прочитать о них.

Управление указателями на массивы, которые не покидают область видимости

Мы уже видели, как управлять указателями на ресурсы, в рецепте «*Управление указателями на классы, которые не покидают область видимости*». Но когда мы имеем дело с массивами, нам нужно вызвать delete[] вместо простого delete. В противном случае произойдет утечка памяти. Посмотрите на приведенный ниже код:

```
void may_throw1(char ch);
void may_throw2(const char* buffer);

void foo() {
    // Мы не можем выделить 10 МБ памяти в стеке, поэтому выделяем ее в куче
    char* buffer = new char[1024 * 1024 * 10];

    // Ой. Этот код может выбросить исключение.
    // Это была плохая идея использовать обычный  указатель, поскольку может произойти
    // утечка памяти!
    may_throw1(buffer[0]);
    may_throw2(buffer);

    delete[] buffer;
}
```

Подготовка

Для этого рецепта требуется знание C++-исключений и шаблонов.

Как это делается...

В библиотеке Boost.SmartPointer есть не только класс scoped_ptr<>, но и класс scoped_array<>:

```
#include <boost/scoped_array.hpp>

void foo_fixed() {
    // Выделяем массив в куче
    boost::scoped_array<char> buffer(new char[1024 * 1024 * 10]);

    // Этот код может выбросить исключение, но теперь оно не приведет к утечке памяти.
    may_throw1(buffer[0]);
    may_throw2(buffer.get());

    // деструктор переменной 'buffer' вызовет delete[]
}
```

Класс boost::movelib::unique_ptr<> библиотеки Boost.Move также может работать с массивами. Вам просто нужно указать, что он хранит массив, поставив в конце параметра шаблона квадратные скобки []:

```
#include <boost/move/make_unique.hpp>

void foo_fixed2() {
    // Выделяем массив в куче
    const boost::movelib::unique_ptr<char[]> buffer
        = boost::movelib::make_unique<char[]>(1024 * 1024 * 10);
    // Этот код может выбросить исключение, но теперь оно не приведет к утечке памяти.
    may_throw1(buffer[0]);
    may_throw2(buffer.get());

    // деструктор переменной 'buffer' вызовет delete[]
}
```

Как это работает...

Класс scoped_array<> работает точно так же, как и класс scoped_ptr<>, но вызывает delete[] в деструкторе вместо delete. Класс unique_ptr<T[]> делает то же самое.

Дополнительно...

Класс scoped_array<> имеет тот же дизайн, что и класс scoped_ptr<>. Он не выделяет память и не вызывает виртуальные функции. Его нельзя скопировать, и он также не является частью C++11. std::unique_ptr<T[]> является частью C++11, и у него та же производительность, что и у класса boost::movelib::unique_ptr<T[]>.

На самом деле make_unique<char[]>(1024) и new char[1024] – не одно и то же, потому что первый выполняет инициализацию значениями (value initialization), а второй – инициализацию по умолчанию (default initialization). Чтобы вызвать иницилизацию по умолчанию для элементов boost::movelib::unique_ptr<T[]>, используйте функцию boost::movelib::make_unique_definit.

Обратите внимание, что Boost-версия может также работать со старыми стандартами C++, предшествующими C++11, и даже эмулировать в них rvalue-ссылки, делая `boost::movelib::unique_ptr` типом, который можно перемещать, но нельзя копировать.

Если ваша стандартная библиотека не предоставляет `std::make_unique`, вам может помочь библиотека `Boost.SmartPtr`. Она предоставляет `boost::make_unique`, который возвращает `std::unique_ptr` в заголовке `boost/smart_ptr/make_unique.hpp`, а также предоставляет `boost::make_unique_noinit` для инициализации по умолчанию в том же заголовке. В C++17 нет функции `make_unique_noinit`.

Использование `new` для выделения памяти и ручного управления памятью в C++ – плохая привычка. Используйте функции `make_unique` и `make_shared` везде, где это возможно.

См. также

- В документации по библиотеке `Boost.SmartPtr` содержится множество примеров и другая полезная информация обо всех классах умных указателей, о которых можно прочитать на странице http://boost.org/libs/smart_ptr;
- документы `Boost.Move` могут помочь вам, если вы хотите использовать эмуляцию перемещения с `boost::movelib::unique_ptr`. Читайте о них по адресу http://boost.org/libs/move.

Подсчет указателей на массивы

Мы продолжаем работать с указателями, и наша следующая задача – подсчитать указатели на массив. Давайте посмотрим на программу, которая получает данные и обрабатывает их в разных потоках. Код для этого выглядит следующим образом:

```
#include <cstring>
#include <boost/thread.hpp>
#include <boost/bind.hpp>

void do_process(const char* data, std::size_t size);

void do_process_in_background(const char* data, std::size_t size) {
    // Нам нужно скопировать данные, потому что мы не знаем, когда они будут освобождены
    // вызывающим кодом.
    char* data_cpy = new char[size];
    std::memcpy(data_cpy, data, size);

    // Запускаем потоки выполнения для обработки данных.
    boost::thread(boost::bind(&do_process, data_cpy, size))
            .detach();
    boost::thread(boost::bind(&do_process, data_cpy, size))
            .detach();

    // Ой!!! Мы не можем вызвать delete[] data_cpy, потому что функция do_process()
    // все еще может работать с data_cpy.
}
```

Точно такая же проблема возникла в рецепте «*Подсчет указателей на классы*».

Подготовка

В этом рецепте используется библиотека Boost.Thread, которая не является библиотекой header-only, поэтому вашей программе потребуется линковаться с boost_thread, boost_chrono и boost_system. Убедитесь, что вы понимаете концепцию потоков, прежде чем читать дальше.

Вам также понадобятся базовые знания по boost::bind или std::bind, что практически одно и то же.

Как это делается...

Есть четыре решения. Основное различие между ними заключается в типе и создании переменной data_cpy. Все эти решения делают то же самое, что и пример в начале этого рецепта, но без утечек памяти. Вот какие возможны решения:

- первое решение хорошо подходит для случаев, когда размер массива известен во время компиляции:

```
#include <boost/shared_ptr.hpp>
#include <boost/make_shared.hpp>

template <std::size_t Size>
void do_process_shared(const boost::shared_ptr<char[Size]>& data);

template <std::size_t Size>
void do_process_in_background_v1(const char* data) {
      // Та же скорость, что и в первом решении.
      boost::shared_ptr<char[Size]> data_cpy
          = boost::make_shared<char[Size]>();
      std::memcpy(data_cpy.get(), data, Size);

      // Запускаем потоки выполнения для обработки данных.
      boost::thread(boost::bind(&do_process_shared<Size>, data_cpy))
          .detach();
      boost::thread(boost::bind(&do_process_shared<Size>, data_cpy))
          .detach();

      // Деструктор data_cpy освободит данные, когда счетчик ссылок будет равен нулю.
}
```

- начиная с Boost версии 1.53 shared_ptr может сам позаботиться о массивах низвестного размера. Второе решение:

```
#include <boost/shared_ptr.hpp>
#include <boost/make_shared.hpp>

void do_process_shared_ptr(
        const boost::shared_ptr<char[]>& data,
        std::size_t size);

void do_process_in_background_v2(const char* data, std::size_t size) {
      // Быстрее чем в первом решении.
      boost::shared_ptr<char[]> data_cpy
          = boost::make_shared<char[]>(size);
      std::memcpy(data_cpy.get(), data, size);
```

```
    // Запускаем потоки выполнения для обработки данных.
    boost::thread(boost::bind(&do_process_shared_ptr, data_cpy, size))
        .detach();
    boost::thread(boost::bind(&do_process_shared_ptr, data_cpy, size))
        .detach();

    // Деструктор data_cpy освободит данные, когда счетчик ссылок будет равен нулю.
}
```

- третье решение:

```
#include <boost/shared_ptr.hpp>

void do_process_shared_ptr2(
        const boost::shared_ptr<char>& data,
        std::size_t size);

void do_process_in_background_v3(const char* data, std::size_t size) {
    // Производительность аналогична первому решению.'.
    boost::shared_ptr<char> data_cpy(
                new char[size],
                boost::checked_array_deleter<char>()
    );
    std::memcpy(data_cpy.get(), data, size);

    // Запускаем потоки выполнения для обработки данных
    boost::thread(boost::bind(&do_process_shared_ptr2, data_cpy, size))
         .detach();
    boost::thread(boost::bind(&do_process_shared_ptr2, data_cpy, size))
         .detach();

    // Деструктор data_cpy освободит данные, когда счетчик ссылок будет равен нулю.
}
```

- последнее решение устарело после появления Boost версии 1.65, но может быть полезно в случае старых версий Boost:

```
#include <boost/shared_array.hpp>

void do_process_shared_array(
        const boost::shared_array<char>& data,
        std::size_t size);
void do_process_in_background_v4(const char* data, std::size_t size) {
    // Нам нужно скопировать данные, потому что мы не знаем, когда они будут
    // освобождены вызывающим кодом.
    boost::shared_array<char> data_cpy(new char[size]);
    std::memcpy(data_cpy.get(), data, size);

    // Запускаем потоки выполнения для обработки данных
    boost::thread(
        boost::bind(&do_process_shared_array, data_cpy, size)
    ).detach();
    boost::thread(
        boost::bind(&do_process_shared_array, data_cpy, size)
```

```
        ).detach();

        // Нет необходимости вызывать delete[] для data_cpy, потому что деструктор data_cpy
        // освободит данные, когда счетчик ссылок будет равен нулю
    }
```

Как это работает...

Во всех примерах классы **умных указателей** подсчитывают ссылки и вызывают `delete[]` для указателя, когда количество ссылок становится равным нулю. Первый и второй примеры просты. В третьем примере мы предоставляем пользовательский объект `deleter`. Объект `deleter` умного указателя вызывается, когда указатель решает освободить ресурсы. Когда умный указатель создается без явного объекта `deleter`, используется `deleter` по умолчанию, который вызывает `delete` или `delete[]` в зависимости от шаблонного параметра указателя.

Дополнительно...

Четвертое решение является наиболее консервативным, поскольку до появления Boost версии 1.53 функциональность второго решения не была реализована в `shared_ptr`. Первое и второе решения являются самыми быстрыми, поскольку они выделяют память только единожды.

Третье решение можно использовать с более старыми версиями Boost и с `std::shared_ptr<>` из стандартной библиотеки C++11(просто не забудьте изменить `boost::checked_array_deleter<T>()` на `std::default_delete <T[]>()`).

На самом деле `boost::make_shared<char []>(size)` и `new char[size]` – не одно и то же, потому что он включает в себя инициализацию значения всех элементов. Эквивалентная функция для инициализации по умолчанию – это `boost::make_shared_noinit`.

Осторожно! Версии `std::shared_ptr` для C++11 и C++14 не могут работать с массивами!

Только начиная с C++17 `std::shared_ptr<T[]>` должна работать правильно. Если вы планируете писать переносимый код, рассмотрите возможность использования `boost::shared_ptr`, `boost::shared_array` или явной передачи объекта `deleter` в `std::shared_ptr`.

`boost::shared_ptr <T[]>`, `boost::shared_array` и `std::shared_ptr<T[]>` из C++17 имеют `operator[](std::size_t index)`, позволяющий получить доступ к элементам совместно используемого массива по индексу. `boost::shared_ptr<T>` и `std::shared_ptr<T>` с пользовательским объектом `deleter` не имеют оператора [], что делает их менее полезными.

См. также

В документации по библиотеке `Boost.SmartPtr` содержится множество примеров и другая полезная информация обо всех классах умных указателей. Вы можете прочитать об этом на странице http://boost.org/libs/smart_ptr.

Хранение любых функциональных объектов в переменной

Рассмотрим ситуацию, когда вы разрабатываете библиотеку, API которой объявлен в заголовочных файлах, а реализация – в исходных файлах. Вам надо написать функцию, которая принимает любые функциональные объекты. Посмотрите на приведенный ниже код:

```
// Создание typedef для указателя на функцию, принимающего int и ничего не возвращающего
typedef void (*func_t)(int);

// Функция, которая принимает указатель на функцию и вызывает принятую функцию для
// каждого целого числа, которое у нее есть. Она не может работать с функциональными
// объектами :(
void process_integers(func_t f);

// Функциональный объект
class int_processor {
   const int min_;
   const int max_;
   bool& triggered_;

public:
    int_processor(int min, int max, bool& triggered)
        : min_(min)
        , max_(max)
        , triggered_(triggered)
    {}
    void operator()(int i) const {
        if (i < min_ || i > max_) {
            triggered_ = true;
        }
     }
};
```

Как изменить функцию `process_integers`, чтобы принимать любые функциональные объекты?

Подготовка

Перед началом работы с этим рецептом рекомендуется прочитать рецепт «*Хранение любого значения в контейнере или переменной*» главы 1 «*Приступаем к написанию приложения*».

Как это делается...

Решение есть, и это – библиотека `Boost.Function`. Она позволяет хранить любую функцию, функцию-член (member function) или функциональный объект, если его сигнатура совпадает с сигнатурой, описанной в аргументе шаблона:

```
#include <boost/function.hpp>

typedef boost::function<void(int)> fobject_t;
```

```
// Теперь эта функция может принимать функциональные объекты
void process_integers(const fobject_t& f);

int main() {
    bool is_triggered = false;
    int_processor fo(0, 200, is_triggered);
    process_integers(fo);
    assert(is_triggered);
}
```

Как это работает...

Объект `fobject_t` хранит в себе функциональные объекты и «стирает» информацию об их типе. Можно использовать класс `boost::function` для объектов с состоянием:

```
bool g_is_triggered = false;
void set_functional_object(fobject_t& f) {
    // Локальная переменная
    int_processor fo(100, 200, g_is_triggered);

    f = fo;
    // теперь 'f' содержит копию 'fo'

    // 'fo' выходит за пределы области видимости и будет уничтожен, но можно
    // использовать 'f' во внешней области видимости.
}
```

`boost::function` не напоминает вам класс `boost::any`? Это потому, что он использует тот же метод **стирания типов** для хранения любых функциональных объектов.

Дополнительно...

У класса `boost::function` есть конструктор по умолчанию, инициализирующий объекты в пустое состояние. Проверку на пустое состояние можно сделать следующим образом:

```
void foo(const fobject_t& f) {
    // Класс boost::function конвертируется в bool
    if (f) {
        // у нас есть значение в 'f'
        // ...
    } else {
        // 'f' пустое
        // ...
    }
}
```

Библиотека `Boost.Function` имеет безумное количество оптимизаций. Она может хранить небольшие функциональные объекты без дополнительного выделения памяти, и у нее есть операторы перемещения. Она принята как часть

стандартной библиотеки C++11 и определяется в заголовке `<functional>` в пространстве имен `std::`.

Класс `boost::function` использует динамическую идентификацию типа данных (RTTI) для объектов, хранящихся в нем. Если вы отключите динамическую идентификацию, библиотека продолжит работать, но значительно увеличит размер скомпилированного двоичного файла.

См. также

- В официальной документации по библиотеке `Boost.Function` содержатся дополнительные примеры, показатели производительности и справочная документации по классам. Перейдите по ссылке http://boost.org/libs/function, чтобы прочитать об этом;
- рецепт «*Передача указателя на функцию в переменной*»;
- рецепт «*Передача лямбда-функций C++11 в переменной*».

Передача указателя на функцию

Мы продолжаем работать с предыдущим примером и теперь хотим передать указатель на функцию в нашем методе `process_integers()`. Должны ли мы добавить перегрузку только для указателей на функции или есть более элегантный способ?

Подготовка

Этот рецепт является продолжением предыдущего. Вначале вы должны прочитать предыдущий рецепт.

Как это делается...

Ничего не нужно делать, поскольку `boost::function<>` также конструируется из указателей на функции:

```
void my_ints_function(int i);

int main() {
    process_integers(&my_ints_function);
}
```

Как это работает...

Указатель на `my_ints_function` будет храниться в классе `boost::function`, а вызовы этого класса будут перенаправляться на сохраненный указатель.

Дополнительно...

Библиотека `Boost.Function` обеспечивает хорошую производительность для указателей на функции и не выделяет память в куче. Стандартная библиотека `std::function` также эффективна для хранения указателей на функции. Начиная с Boost версии 1.58 библиотека `Boost.Function` может хранить функции и функциональные объекты, имеющие сигнатуру вызова с rvalue-ссылками:

```
boost::function<int(std::string&&)> f = &something;
f(std::string("Hello")); // Работает
```

См. также

- В официальной документации по библиотеке `Boost.Function` содержатся дополнительные примеры, показатели производительности и справочная документации по классам. Перейдите по ссылке http://boost.org/libs/function, чтобы прочитать об этом;
- рецепт «*Передача лямбда-функций C++11 в переменной*».

Хранение любых лямбда-функций C++11 в переменной

Мы продолжаем работать с предыдущим примером и теперь хотим использовать лямбда-функцию с нашим методом `process_integers()`.

Подготовка

Этот рецепт продолжает серию предыдущих двух. Вначале вы должны прочитать их. Вам также понадобится компилятор с C++11 или хотя бы компилятор с поддержкой лямбда-выражений из C++11.

Как это делается...

Ничего не нужно делать с примером из предыдущего рецепта, поскольку `boost::function<>` также можно использовать с лямбда-функциями любой сложности:

```
#include <deque>
//#include "your_project/process_integers.h"

void sample() {
    // лямбда-функция без параметров, которая ничего не делает
    process_integers([](int /*i*/){});

    // лямбда-функция, которая хранит ссылку
    std::deque<int> ints;
    process_integers([&ints](int i){
        ints.push_back(i);
    });

    // лямбда-функция, которая изменяет свое содержимое
    std::size_t match_count = 0;
    process_integers([ints, &match_count](int i) mutable {
        if (ints.front() == i) {
           ++ match_count;
        }
        ints.pop_front();
    });
}
```

Дополнительно...

Производительность хранения лямбда-функции в `Boost.Functional` такая же, как и в случаях хранения функциональных объектов. Пока функциональный

объект, созданный лямбда-выражением, достаточно мал, чтобы поместиться в экземпляр класса `boost::function`, динамическое выделение памяти не происходит. Вызов объекта, хранящегося в классе `boost::function`, близок к скорости вызова функции по указателю. При копировании экземпляров класса `boost::function` память в куче выделяется, только если исходный `boost::function` содержит объект, который размещен в куче. Перемещение экземпляров не выделяет и не освобождает память.

Помните, что `boost::function` является оптимизационным барьером для компилятора. Это означает, что

```
std::for_each(v.begin(), v.end(), [](int& v) { v += 10; });
```

Обычно лучше оптимизируется компилятором, чем

```
const boost::function<void(int&)> f0(
    [](int& v) { v += 10; }
);
std::for_each(v.begin(), v.end(), f0);
```

Вот почему нужно стараться избегать использования библиотеки `Boost.Function`, когда ее применение на самом деле не требуется. В некоторых случаях ключевое слово `auto` в C++11 может оказаться более полезным, чем `boost::function`:

```
const auto f1 = [](int& v) { v += 10; };
std::for_each(v.begin(), v.end(), f1);
```

См.также

Дополнительную информацию о производительности и библиотеке `Boost.Function` можно найти на официальной странице документации по адресу http://www.boost.org/libs/function.

Контейнеры указателей

Бывают случаи, когда нам нужно хранить указатели в контейнере. Например: хранение полиморфных данных в контейнерах, для более быстрого копирования данных в контейнерах. В таких случаях у программиста C++ есть следующие варианты:

- хранить указатели в контейнерах и избавляться от них, используя оператор `delete`:

```
#include <set>
#include <algorithm>
#include <cassert>

template <class T>
struct ptr_cmp {
    template <class T1>
    bool operator()(const T1& v1, const T1& v2) const {
        return operator ()(*v1, *v2);
    }
```

```
        bool operator()(const T& v1, const T& v2) const {
            return std::less<T>()(v1, v2);
        }
    };

    void example1() {
        std::set<int*, ptr_cmp<int> > s;
        s.insert(new int(1));
        s.insert(new int(0));

        // ...
        assert(**s.begin() == 0);
        // ...

        // Ой! Любое исключение в приведенном выше коде приводит к утечке памяти.

        // Освобождение ресурсов.
        std::for_each(s.begin(), s.end(), [](int* p) { delete p; });
    }
```

Такой подход подвержен ошибкам и требует написания большого количества кода;

- хранить умные указатели C++11 в контейнерах:

```
    #include <memory>
    #include <set>

    void example2_cpp11() {
        typedef std::unique_ptr<int> int_uptr_t;
        std::set<int_uptr_t, ptr_cmp<int> > s;
        s.insert(int_uptr_t(new int(1)));
        s.insert(int_uptr_t(new int(0)));

        // ...
        assert(**s.begin() == 0);
        // ...

        // Ресурсы будут освобождены unique_ptr<>.
    }
```

Это хорошее решение, но его нельзя использовать в C++03, и вам все равно нужно писать компаратор.

В C++14 есть функция `std::make_unique` для построения `std::uniue_ptr`. Использовать ее вместо `new` – хороший стиль кодирования!

- использовать библиотеку `Boost.SmartPtr` в контейнере:

```
#include <boost/shared_ptr.hpp>
#include <boost/make_shared.hpp>

void example3() {
    typedef boost::shared_ptr<int> int_sptr_t;
    std::set<int_sptr_t, ptr_cmp<int> > s;
    s.insert(boost::make_shared<int>(1));
```

```
    s.insert(boost::make_shared<int>(0));

    // ...
    assert(**s.begin() == 0);
    // ...

    // Ресурсы будут освобождены с помощью shared_ptr<>.
}
```

Это решение переносимо, но с ним мы теряем производительность и нерационально используем ресурсы системы (атомарному счетчику требуется дополнительная память, и его увеличение или уменьшение не такое быстрое, как неатомарные операции). И вам все равно нужно писать компараторы.

Подготовка

Для лучшего понимания этого рецепта требуется знание контейнеров стандартной библиотеки.

Как это делается...

Библиотека `Boost.PointerContainer` предоставляет хорошее и портативное решение:

```
#include <boost/ptr_container/ptr_set.hpp>

void correct_impl() {
    boost::ptr_set<int> s;
    s.insert(new int(1));
    s.insert(new int(0));

    // ...
    assert(*s.begin() == 0);
    // ...

    // Ресурсы будут освобождены самим контейнером.
}
```

Как это работает...

В библиотеке `Boost.PointerContainer` есть классы `ptr_array`, `ptr_vector`, `ptr_set`, `ptr_multimap` и др. Эти классы освобождают указатели по мере необходимости и упрощают доступ к данным, на которые ссылается указатель (нет необходимости в дополнительном разыменовании в `assert(* s.begin () == 0);`).

Дополнительно...

Когда мы хотим клонировать какие-то данные, нам нужно определить функцию `T* new_clone(const T& r)` в пространстве имен клонируемого объекта. Кроме того, вы можете использовать реализацию `T * new_clone(const T& r)` по умолчанию, если подключите заголовочный файл `<boost/ptr_container/clone_allocator.hpp>`, как показано в приведенном ниже коде:

```
#include <boost/ptr_container/clone_allocator.hpp>
#include <boost/ptr_container/ptr_vector.hpp>
```

```
#include <cassert>

void theres_more_example() {
    // Создаем вектор из 10 элементов со значением 100
    boost::ptr_vector<int> v;
    int value = 100;
    v.resize(10, &value); // Осторожно! Нет владения указателем!

    assert(v.size() == 10);
    assert(v.back() == 100);
}
```

В стандартной библиотеке C++ нет контейнеров указателей, но можно добиться похожей функциональности, используя контейнер и `std::unique_ptr`. Кстати, начиная с Boost версии 1.58 есть класс `boost::movelib::unique_ptr`, который можно использовать в C++03. Вы можете совмещать его с контейнерами из библиотеки `Boost.Container`, чтобы иметь функциональность C++11 для хранения указателей:

```
#include <boost/container/set.hpp>
#include <boost/move/make_unique.hpp>
#include <cassert>

void example2_cpp03() {
    typedef boost::movelib::unique_ptr<int> int_uptr_t;
    boost::container::set<int_uptr_t, ptr_cmp<int> > s;
    s.insert(boost::movelib::make_unique<int>(1));
    s.insert(boost::movelib::make_unique<int>(0));

    // ...
    assert(**s.begin() == 0);
}
```

Не все разработчики хорошо знают библиотеки Boost. Более удобным для большинства разработчиков является использование функций и классов, которые имеют альтернативы в стандартной библиотеке C++, поскольку обычно разработчики хорошо осведомлены о функциях стандартной библиотеки. Поэтому, если для вас нет большой разницы, используйте библиотеку `Boost.Container` с классом `boost::movelib::unique_ptr`.

См. также

- В официальной документации содержится подробная ссылка для каждого класса. Перейдите на страницу http://boost.org/libs/ptr_container, чтобы прочитать об этом;
- в первых четырех рецептах из этой главы приводятся примеры использования умных указателей;
- в нескольких рецептах из главы 9 «Контейнеры» описываются функции библиотеки `Boost.Container`. Просмотрите эту главу, чтобы найти для себя отличные, полезные и быстрые контейнеры.

Делайте это при выходе из области видимости!

Если вы имели дело с такими языками, как Java, C# или Delphi, то, очевидно, использовали конструкцию `try {} finally{}`. Позвольте мне кратко описать вам, что делают эти языковые конструкции.

Когда выполнение программы выходит из текущей области видимости с помощью `return` или выброса исключения, выполняется код в блоке `finally`. Этот механизм используется в качестве замены для C++ идиомы RAII:

```
// Какой-то псевдокод (подозрительно похожий на код Java):
try {
    FileWriter f = new FileWriter("example_file.txt");
    // Некий код, который может выбросить исключение или воспользоваться return
    // ...
} finally {
    // Что бы ни случилось выше в области видимости, этот код будет выполнен,
    // и файл будет правильно закрыт
    if (f != null) {
        f.close()
    }
}
```

Можно ли сделать такое в C++?

Подготовка

Для этого рецепта требуются базовые знания C++.

Как это делается...

C++ использует идиому проектирования RAII вместо конструкции `try {} finally{}`. Библиотека `Boost.ScopeExit` была разработана для того, чтобы пользователь мог определять RAII-обертки прямо в теле функции:

```
#include <boost/scope_exit.hpp>
#include <cstdlib>
#include <cstdio>
#include <cassert>

int main() {
    std::FILE* f = std::fopen("example_file.txt", "w");
    assert(f);

    BOOST_SCOPE_EXIT(f) {
       // Что бы ни случилось во внешней области видимости, этот код будет
       // выполнен, и файл будет закрыт правильно.
       std::fclose(f);
    } BOOST_SCOPE_EXIT_END

    // Некий код, который может выбросить исключение или воспользоваться return
    // ...
}
```

Как это работает...

f передается по значению в BOOST_SCOPE_EXIT(f). Когда программа покидает область видимости, выполняется код между BOOST_SCOPE_EXIT(f) { и } BOOST_SCOPE_EXIT_END. Если вы хотите передать значение по ссылке, используйте символ & в макросе BOOST_SCOPE_EXIT. Если нужно передать несколько значений, просто разделите их запятыми.

Передача ссылок на указатель в некоторых компиляторах не работает для макроса BOOST_SCOPE_EXIT (&f). Поэтому в примере мы и не захватываем f по ссылке.

Дополнительно...

Чтобы захватить this внутри функции-члена, мы будем использовать специальный символ this_:

```
class theres_more_example {
public:
    void close(std::FILE*);

    void theres_more_example_func() {
        std::FILE* f = 0;
        BOOST_SCOPE_EXIT(f, this_) { // Захватываем `this` как 'this_'
            this_->close(f);
        } BOOST_SCOPE_EXIT_END
    }
};
```

Библиотека Boost.ScopeExit не выделяет дополнительной памяти в куче и не использует виртуальные функции. Используйте синтаксис по умолчанию и не определяйте BOOST_SCOPE_EXIT_CONFIG_USE_LAMBDAS, потому что в противном случае выход из области будет реализован с использованием класса boost::function, что может выделить дополнительную память и создать барьер для оптимизации. Можно добиться близких к BOOST_SCOPE_EXIT результатов, используя boost::movelib::unique_ptr или std::unique_ptr путем указания пользовательского объекта deleter:

```
#include <boost/move/unique_ptr.hpp>
#include <cstdio>

void unique_ptr_example() {
    boost::movelib::unique_ptr<std::FILE, int(*)(std::FILE*)> f(
        std::fopen("example_file.txt", "w"), // открыть файл
        &std::fclose // пользовательский deleter
    );
    // ...
}
```

Если вы пишете два или более одинаковых тела для BOOST_SCOPE_EXIT, то пришло время подумать о рефакторинге и переносе кода в RAII-класс.

См. также

В официальной документации содержится больше примеров и вариантов использования. Вы можете прочитать об этом на странице http://boost.org/libs/scope_exit.

Инициализация базового класса членом класса-наследника

Посмотрим на приведенный ниже пример. У нас есть некий базовый класс, у которого имеются виртуальные функции и который должен быть проинициализирован ссылкой на объект `std::ostream`:

```
#include <boost/noncopyable.hpp>
#include <sstream>

class tasks_processor: boost::noncopyable {
    std::ostream& log_;

protected:
    virtual void do_process() = 0;

public:
    explicit tasks_processor(std::ostream& log)
        : log_(log)
    {}

    void process() {
        log_ << "Starting data processing";
        do_process();
    }
};
```

У нас также есть производный класс, у которого имеется объект `std::ostream` и который реализует функцию `do_process()`:

```
class fake_tasks_processor: public tasks_processor {
    std::ostringstream logger_;

    virtual void do_process() {
        logger_ << "Fake processor processed!";
    }

public:
    fake_tasks_processor()
        : tasks_processor(logger_) // Ой! logger_ еще не создан
        , logger_()
     {}
};
```

Это не очень распространенный случай в программировании, но когда такие ситуации случаются, не всегда бывает просто сообразить, как их обойти. Некоторые пытаются сделать это путем изменения порядка `logger_` и базового типа в списке инициализации:

```
fake_tasks_processor()
    : logger_() // Ой! Он по-прежнему создается ПОСЛЕ task_processor.
```

```
    , tasks_processor(logger_)
{}
```

Этот кусок кода не будет работать как ожидалось, потому что базовые классы инициализируются перед нестатическими членами-данными, независимо от их порядка в списке инициализации.

Подготовка

Для этого рецепта требуются базовые знания C++.

Как это делается...

Библиотека `Boost.Utility` предоставляет быстрое решение для таких случаев. Это шаблон `boost::base_from_member`. Чтобы использовать его, необходимо выполнить следующие шаги:

1. Подключите заголовочный файл `base_from_member.hpp`:

```
#include <boost/utility/base_from_member.hpp>
```

2. Унаследуйте свой класс от `boost::base_from_member<T>`, где `T` – это тип, который должен быть проинициализирован перед инициализацией базового класса (позаботьтесь о порядке базовых классов; `boost::base_from_member<T>` должен быть помещен перед классом, который использует `T`):

```
class fake_tasks_processor_fixed
    : boost::base_from_member<std::ostringstream>
    , public tasks_processor
```

3. Напишите конструктор:

```
{
   typedef boost::base_from_member<std::ostringstream> logger_t;

   virtual void do_process() {
       logger_t::member << "Fake processor processed!";
   }

public:
    fake_tasks_processor_fixed()
        : logger_t()
        , tasks_processor(logger_t::member)
    {}
};
```

Как это работает...

Базовые классы инициализируются перед нестатическими членами в порядке их объявления в списке базовых классов. Если нам нужно инициализировать базовый класс `B` *чем-то*, нам нужно сделать это *что-то* частью базового класса `A`, который объявлен до `B`. Другими словами, `boost::base_from_member` – это лишь простой класс, который содержит параметр шаблона в качестве нестатического члена-данных:

```
template < typename MemberType, int UniqueID = 0 >
class base_from_member {
protected:
    MemberType member;
    // Здесь идут конструкторы...
};
```

Дополнительно...

Как видите, base_from_member имеет целое число в качестве второго аргумента шаблона. Это делается для тех случаев, когда нам нужно несколько классов base_from_member одного типа:

```
class fake_tasks_processor2
    : boost::base_from_member<std::ostringstream, 0>
    , boost::base_from_member<std::ostringstream, 1>
    , public tasks_processor
{
    typedef boost::base_from_member<std::ostringstream, 0> logger0_t;
    typedef boost::base_from_member<std::ostringstream, 1> logger1_t;
    virtual void do_process() {
        logger0_t::member << "0: Fake processor2 processed!";
        logger1_t::member << "1: Fake processor2 processed!";
    }

public:
    fake_tasks_processor2()
        : logger0_t()
        , logger1_t()
        , tasks_processor(logger0_t::member)
    {}
};
```

Класс boost::base_from_member не использует динамическое выделение памяти, и у него нет виртуальных функций. Текущая реализация поддерживает **прямую передачу (perfect forwarding)** и **вариативные шаблоны (variadic templates)**, если их поддерживает ваш компилятор.

В стандартной библиотеке C++ нет base_from_member.

См. также

- Библиотека Boost.Utility содержит множество полезных классов и функций; документация для получения дополнительной информации находится на странице http://boost.org/libs/utility;
- в рецепте «*Создание некопируемого класса*» главы 1 «Приступаем к написанию приложения» содержатся дополнительные примеры классов из библиотеки Boost.Utility;
- кроме того, в рецепте «*Использование эмуляции перемещения в C++11*» главы 1 «Приступаем к написанию приложения» также содержатся дополнительные примеры классов из библиотеки Boost.Utility.

Глава 3

Преобразование и приведение

Темы, которые мы рассмотрим в этой главе:

- преобразование строк в числа;
- преобразование чисел в строки;
- преобразование чисел в числа;
- преобразование пользовательских типов в строки и из строк;
- приведение умных указателей;
- приведение полиморфных объектов;
- синтаксический анализ (parsing) простого ввода;
- синтаксический анализ (parsing) сложного ввода.

Вступление

Теперь, когда мы познакомились с базовыми типами Boost, пришло время познакомиться с функциями преобразования данных. В этой главе мы увидим, как преобразовывать строки, числа, указатели и пользовательские типы друг в друга, как безопасно приводить полиморфные типы и как писать маленькие и большие парсеры прямо в исходных файлах C++.

Преобразование строк в числа

Преобразование строк в числа в C++ вызывает у многих депрессию из-за своей неэффективности и недружелюбности по отношению к пользователям. Посмотрите, как можно преобразовать строку "100" в `int`:

```
#include <sstream>

void sample1() {
    std::istringstream iss("100");
    int i;
    iss >> i;

    // ...
}
```

Лучше не думать, сколько ненужных операций, вызовов виртуальных функций, атомарных операций и выделений памяти произошло во время этого преобразования. Кстати, нам больше не нужна переменная `iss`, но она будет жить до конца области видимости.

Методы языка C не намного лучше:

```
#include <cstdlib>

void sample2() {
    char * end;
    const int i = std::strtol ("100", &end, 10);

    // ...
}
```

Было ли преобразовано все значение в `int` или остановили разбор где-то посередине? Чтобы понять это, мы должны проверить содержимое переменной `end`. После этого у нас будет бесполезная переменная `end`, которая будет путаться под ногами до конца области видимости. При этом мы хотели получить `int`, а `strtol` возвращает `long int`. Можно ли сохранить преобразованное значение в `int` без потери значащих разрядов?

Подготовка

Для этого рецепта требуются только базовые знания C++ и стандартной библиотеки.

Как это делается...

В Boost есть библиотека, которая поможет вам справиться с трудностями преобразования строк в числа. Она называется `Boost.LexicalCast` и состоит из класса-исключения `boost::bad_lexical_cast`, функций `boost::lexical_cast` и `boost::conversion::try_lexical_convert`:

```
#include <boost/lexical_cast.hpp>

void sample3() {
    const int i = boost::lexical_cast<int>("100");
     // ...
}
```

Ее даже можно использовать для строк, которые не являются нуль-терминированными:

```
#include <boost/lexical_cast.hpp>

void sample4() {
    char chars[] = {'x', '1', '0', '0', 'y' };
    const int i = boost::lexical_cast<int>(chars + 1, 3);
    assert(i == 100);
}
```

Как это работает...

Функция `boost::lexical_cast` принимает строку в качестве входных данных и преобразует ее в тип, указанный в треугольных скобках. Она даже проверит за вас границы:

```
#include <boost/lexical_cast.hpp>
#include <iostream>

void sample5() {
   try {
       // обычно short не может хранить значения больше 32767
       const short s = boost::lexical_cast<short>("1000000");
       assert(false); // Must not reach this line.
  } catch (const boost::bad_lexical_cast& e) {
      std::cout << e.what() << '\n';
  }
}
```

Предыдущий код выводит:

```
bad lexical cast: source type value could not be interpreted as target.
```

Он также проверяет правильность синтаксиса ввода и выбрасывает исключение, если ввод неверный:

```
#include <boost/lexical_cast.hpp>
#include <iostream>

void sample6() {
   try {
       const int i = boost::lexical_cast<int>("This is not a number!");
       assert(false); // Не должен достигать этой строки
    } catch (const boost::bad_lexical_cast& /*e*/) {}
}
```

Начиная с Boost версии 1.56 существует функция `boost::convert::try_lexical_convert`, которая сообщает об ошибках, используя код возврата. Эта функция полезна в критичных к производительности местах, где часто может появляться неправильный ввод:

```
#include <boost/lexical_cast.hpp>
#include <cassert>

void sample7() {
    int i = 0;
    const bool ok = boost::conversion::try_lexical_convert("Bad stuff", i);
    assert(!ok);
}
```

Дополнительно...

`lexical_cast`, как и все классы `std::stringstream`, использует `std::locale` и может преобразовывать локализованные числа, но у него также имеется впечатляющий набор оптимизаций для **C-локали** и локалей без группировок:

```
#include <boost/lexical_cast.hpp>
#include <locale>

void sample8() {
    try {
```

```
        std::locale::global(std::locale("ru_RU.UTF8"));
        // В России знак запятой используется как десятичный разделитель
        float f = boost::lexical_cast<float>("1,0");
        assert(f < 1.01 && f > 0.99);
        std::locale::global(std::locale::classic()); // Восстанавливаем C-локаль
    } catch (const std::runtime_error&) { /* locale is not supported */ }
}
```

В стандартной библиотеке C++ нет шаблонной функции `lexical_cast`, но начиная с C++17 существуют функции `std::from_chars`, которые можно использовать для создания удивительно быстрых универсальных конвертеров. Обратите внимание, что эти конвертеры вообще не используют локали, поэтому они работают несколько иначе, но обеспечивают максимальную производительность. Функции `std::from_chars` не выделяют память, не выбрасывают исключения и не используют атомарных или каких-либо других тяжелых операций.

См. также

- Обратитесь к рецепту «*Преобразование чисел в строки*» для получения информации о производительности функции `boost::lexical_cast`;
- официальная документация `Boost.LexicalCast` содержит примеры, показатели производительности и ответы на часто задаваемые вопросы. Она доступна по адресу http://boost.org/libs/lexical_cast.

Преобразование чисел в строки

В этом рецепте мы продолжим обсуждение лексических преобразований, но теперь будем преобразовывать числа в строки, используя библиотеку `Boost.LexicalCast`. Как обычно, функция `boost::lexical_cast` предоставит очень простой способ конвертации данных.

Подготовка

Для этого рецепта требуются только базовые знания C++ и стандартная библиотека.

Как это делается...

Давайте преобразуем целое число `100` в `std::string`, используя функцию `boost::lexical_cast`:

```
#include <cassert>
#include <boost/lexical_cast.hpp>

void lexical_cast_example() {
    const std::string s = boost::lexical_cast<std::string>(100);
    assert(s == "100");
}
```

Сравните это с традиционным методом преобразования в C++03:

```
#include <cassert>
#include <sstream>

void cpp_convert_example() {
    std::stringstream ss; // Медленный и тяжелый конструктор по умолчанию
    ss << 100;
    std::string s;
    ss >> s;

  // Переменная 'ss' будет маячить здесь до конца области видимости. Во время
  // преобразования было вызвано несколько виртуальных методов и тяжелых операций.
  assert(s == "100");
}
```

с методом преобразования в C:

```
#include <cassert>
#include <cstdlib>

void c_convert_example() {
     char buffer[100];
     std::sprintf(buffer, "%i", 100);

     // Вам понадобится тип unsigned long long int, чтобы подсчитать, сколько раз в мире
     // были допущены ошибки в функциях типа printf. Функции 'printf' являются
     // постоянной угрозой безопасности!

     // Но подождите, нам все еще нужно создать std::string.
     const std::string s = buffer;
     // Теперь у нас есть переменная 'buffer', которая не используется
     assert(s == "100");
}
```

Как это работает...

Функция `boost::lexical_cast` также может принимать числа в качестве входных данных и преобразовывать их в строковый тип, указанный в качестве параметра шаблона (в треугольных скобках). Это очень похоже на то, что мы сделали в предыдущем рецепте.

Дополнительно...

Внимательный читатель заметит, что `lexical_cast` возвращает объект по значению, а значит, у нас есть дополнительный вызов конструктора копирования строки, и что дополнительный вызов снижает производительность. Это верно только для очень старых плохих компиляторов. Современные компиляторы реализуют **оптимизацию именованного возвращаемого значения** (Named Return Value Optimization – **NRVO**), что устраняет ненужные вызовы конструкторов копирования и деструкторов. Даже если компиляторы, совместимые с C++11, не распознают NRVO, они обязаны использовать конструктор перемещения для `std::string`, который быстр и эффективен. В разделе «Производительность» документации `Boost.LexicalCast` показана скорость конвертации разных типов для разных компиляторов. В большинстве случаев `lexical_cast` работает быстрее `std::stringstream` и `printf`.

Если `boost::array` или `std::array` передаются в функцию `boost::lexical_cast` в качестве типа выходного параметра, произойдет меньше динамических аллокаций (или аллокаций не будет вообще, все зависит от реализации `std::locale`).

В C++11 имеются функции `std::to_string` и `std::to_wstring`, которые объявлены в заголовке `<string>`. Эти функции используют локали, и их поведение очень похоже на `boost::lexical_cast<std::string>` и `boost::lexical_cast<std::wstring>` соответственно.

В C++17 есть функции `std::to_chars`, которые удивительно быстро преобразовывают числа в массивы символов. `std::to_chars` не выделяет память, не выбрасывает исключение и может сообщать об ошибках, используя коды ошибок. Если вам нужны действительно быстрые функции преобразования, которые не используют локали, применяйте `std::to_chars`.

См. также

- Официальная документация Boost содержит таблицы, которые сравнивают производительность `lexical_cast` с другими подходами преобразования. В большинстве случаев `lexical_cast` работает быстрее: http://boost.org/libs/lexical_cast;
- рецепт «*Преобразование строк в числа*»;
- рецепт «*Преобразование пользовательских типов в строки и из строк*».

Преобразование чисел в числа

Возможно, вы помните ситуации, когда писали подобный код:

```
void some_function(unsigned short param);
int foo();

void do_something() {
      // Некоторые компиляторы могут предупредить, что int преобразуется в unsigned
      // short и что существует вероятность потери данных.
      some_function(foo());
}
```

Обычно программисты просто игнорируют такие предупреждения компилятора путем явного приведения к нужному типу данных, как показано в приведенном ниже фрагменте кода:

```
// Предупреждение подавлено.
some_function(
    static_cast<unsigned short>(foo())
);
```

Но что, если `foo()` возвращает числа, которые нельзя разместить в `unsigned short`? Это приводит к появлению ошибок, которые крайне трудно обнаружить. Такие ошибки могут существовать в коде годами, прежде чем их обнаружат и исправят:

```
// Возвращает -1, если произошла ошибка.
int foo() {
    if (some_extremely_rare_condition()) {
```

```
        return -1;
    } else if (another_extremely_rare_condition()) {
        return 1000000;
    }
    return 42;
}
```

Подготовка

Для этого рецепта требуются только базовые знания C++.

Как это делается...

Библиотека `Boost.NumericConversion` предоставляет решение для таких случаев. Просто замените `static_cast` на функцию `boost::numeric_cast`. Эта функция выбросит исключение, если исходное значение нельзя сохранить в результирующем типе данных:

```
#include <boost/numeric/conversion/cast.hpp>

void correct_implementation() {
    // На 100 % правильно.
    some_function(
        boost::numeric_cast<unsigned short>(foo())
    );
}

void test_function() {
   for (unsigned int i = 0; i < 100; ++i) {
       try {
           correct_implementation();
       } catch (const boost::numeric::bad_numeric_cast& e) {
           std::cout << '#' << i << ' ' << e.what() << std::endl;
       }
   }
}
```

Теперь, если мы запустим функцию `test_function()`, она выведет следующее:

```
#47 bad numeric conversion: negative overflow
#58 bad numeric conversion: positive overflow
```

Мы даже можем обнаружить определенные типы переполнения:

```
void test_function1() {
   for (unsigned int i = 0; i < 100; ++i) {
       try {
           correct_implementation();
       } catch (const boost::numeric::positive_overflow& e) {
           // Положительное переполнение.
           std::cout << "POS OVERFLOW in #" << i << ' '
                     << e.what() << std::endl;
       } catch (const boost::numeric::negative_overflow& e) {
```

```
            // Отрицательное переполнение.
            std::cout <<"NEG OVERFLOW in #" << i << ' '
                      << e.what() << std::endl;
        }
    }
}
```

Функция test_function1() выведет следующее:

```
NEG OVERFLOW in #47 bad numeric conversion: negative overflow
POS OVERFLOW in #59 bad numeric conversion: positive overflow
```

Как это работает...

Функция boost::numeric_cast проверяет, помещается ли значение входного параметра в результирующий тип без потери данных, и выбрасывает исключение, если во время преобразования что-то переполняется.

Библиотека Boost.NumericConversion имеет очень быструю реализацию. Она выбирает оптимальный алгоритм обнаружения переполнений во время компиляции, например при преобразовании в типы с более широким диапазоном исходное значение будет преобразовано в целевой тип просто через static_cast.

Дополнительно...

Функция boost::numeric_cast реализована в Boost с помощью класса boost::numeric::converter, который можно настроить различными политиками переполнения, проверки диапазона и округления. Но обычно то, что вам нужно, – это numeric_cast.

Вот небольшой пример, который демонстрирует, как создать собственный обработчик переполнения для boost::numeric::cast:

```
template <class SourceT, class TargetT>
struct mythrow_overflow_handler {
    void operator() (boost::numeric::range_check_result r) {
        if (r != boost::numeric::cInRange) {
            throw std::logic_error("Not in range!");
        }
    }
};

template <class TargetT, class SourceT>
TargetT my_numeric_cast(const SourceT& in) {
    typedef boost::numeric::conversion_traits<
        TargetT, SourceT
    > conv_traits;
    typedef boost::numeric::converter <
        TargetT,
        SourceT,
        conv_traits, // default conversion traits
        mythrow_overflow_handler<SourceT, TargetT> // !!!
    > converter;

    return converter::convert(in);
}
```

Вот как использовать наш конвертер:

```
void example_with_my_numeric_cast() {
    short v = 0;
    try {
        v = my_numeric_cast<short>(100000);
    } catch (const std::logic_error& e) {
        std::cout << "It works! " << e.what() << std::endl;
    }
}
```

Данный код выводит следующее сообщение:

```
It works! Not in range!
```

Даже в C++20 нет средств для безопасного числового приведения. Однако работа в этом направлении продолжается. У нас есть все шансы увидеть необходимые методы после 2023 года, в C++23.

См. также

Официальная документация Boost содержит подробное описание всех параметров шаблона числового преобразователя; она доступна по ссылке: http://boost.org/libs/numeric/conversion.

Преобразование пользовательских типов в строки и из строк

В библиотеке `Boost.LexicalCast` есть «фича», позволяющая пользователям применять собственные типы с `lexical_cast`. Чтобы ей воспользоваться, необходимо лишь написать для типа операторы `std::ostream` и `std::istream`.

Как это делается...

1. Все, что вам нужно, – это предоставить потоковые операторы `operator<<` и `operator>>`. Если для вашего класса они уже написаны, то делать ничего не нужно:

```
#include <iostream>
#include <stdexcept>

// Отрицательное число, которое не хранит знак минус
class negative_number {
    unsigned short number_;

public:
    explicit negative_number(unsigned short number = 0)
        : number_(number)
    {}

    // ...
    unsigned short value_without_sign() const {
        return number_;
    }
};
```

```
inline std::ostream&
    operator<<(std::ostream& os, const negative_number& num)
{
    os << '-' << num.value_without_sign();
    return os;
}
inline std::istream& operator>>(std::istream& is, negative_number& num)
{
    char ch;
    is >> ch;
    if (ch != '-') {
        throw std::logic_error(
            "negative_number class stores ONLY negative values"
        );
    }
    unsigned short s;
    is >> s;
    num = negative_number(s);
    return is;
}
```

2. Теперь мы можем использовать функцию `boost::lexical_cast` для конвертации в класс `negative_number` и из него. Например:

```
#include <boost/lexical_cast.hpp>
#include <boost/array.hpp>
#include <cassert>

void example1() {
    const negative_number n
        = boost::lexical_cast<negative_number>("-100");
    assert(n.value_without_sign() == 100);

    const int i = boost::lexical_cast<int>(n);
    assert(i == -100);

    typedef boost::array<char, 10> arr_t;
    const arr_t arr = boost::lexical_cast<arr_t>(n);
    assert(arr[0] == '-');
    assert(arr[1] == '1');
    assert(arr[2] == '0');
    assert(arr[3] == '0');
    assert(arr[4] == 0);
}
```

Как это работает...

Функция `boost::lexical_cast` на этапе компиляции обнаруживает и использует потоковые операторы для преобразования пользовательских типов.

В библиотеке `Boost.LexicalCast` имеется множество оптимизаций для базовых типов, и они будут подхватываться и для пользовательских типов, если пользовательский тип приводится к базовому или если базовый тип приводится к пользовательскому.

Дополнительно...

Функция `boost::lexical_cast` также может работать с Unicode-строками. Для этого требуются правильные перегрузки операторов `basic_istream` и `basic_ostream`:

```
template <class CharT>
std::basic_ostream<CharT>&
    operator<<(std::basic_ostream<CharT>& os, const negative_number& num)
{
    os << static_cast<CharT>('-') << num.value_without_sign();
    return os;

}
template <class CharT>
std::basic_istream<CharT>&
    operator>>(std::basic_istream<CharT>& is, negative_number& num)
{
    CharT ch;
    is >> ch;
    if (ch != static_cast<CharT>('-')) {
        throw std::logic_error(
            "negative_number class stores ONLY negative values"
        );
    }
    unsigned short s;
    is >> s;
    num = negative_number(s);
    return is;
}

void example2() {
    const negative_number n = boost::lexical_cast<negative_number>(L"-1");
    assert(n.value_without_sign() == 1);
    typedef boost::array<wchar_t, 10> warr_t;
    const warr_t arr = boost::lexical_cast<warr_t>(n);
    assert(arr[0] == L'-');
    assert(arr[1] == L'1');
    assert(arr[2] == 0);
}
```

Библиотека `Boost.LexicalCast` не является частью стандарта C++. Многие библиотеки Boost используют ее, и я надеюсь, что она также облегчит и вашу жизнь.

См. также

- Документация по библиотеке `Boost.LexicalCast` содержит примеры, показатели производительности и ответы на часто задаваемые вопросы: http://boost.org/libs/lexical_cast;
- рецепт «*Преобразование строк в числа*»;
- рецепт «*Преобразование чисел в строки*».

Приведение умных указателей

Представьте ситуацию:

1. У вас есть класс с именем base:

```
struct base {
    virtual void some_methods() = 0;
    virtual ~base();
};

struct derived: public base {
    void some_methods() /*override*/;
    virtual void derived_method() const;
    ~derived() /*override*/;
};
```

2. У вас есть сторонний API, который возвращает объект типа derived через умный указатель на base и принимает умный указатель на const derived в других функциях:

```
#include <boost/shared_ptr.hpp>
boost::shared_ptr<const base> construct_derived();
void im_accepting_derived(boost::shared_ptr<const derived> p);
```

3. Вы должны заставить этот код работать:

```
void trying_hard_to_pass_derived() {
   boost::shared_ptr<const base> d = construct_derived();

   // Ой! Ошибка времени компиляции:
   // У ‘const base‘ нет члена с именем derived_method‘.
   d->derived_method();

   // Ой! Ошибка времени компиляции:
   // Не удалось преобразовать ‘d‘ в ‘boost::shared_ptr<const derived>‘.
   im_accepting_derived(d);
}
```

Как решить эту проблему?

Подготовка

Для этого рецепта требуются базовые знания C++ и умных указателей.

Как это делается...

Можно использовать специальные приведения для умных указателей. В данном конкретном случае нам нужно использовать функциональность dynamic_cast, поэтому мы применяем функцию boost::dynamic_pointer_cast:

```
void trying_hard_to_pass_derived2() {
    boost::shared_ptr<const derived> d
        = boost::dynamic_pointer_cast<const derived>(
            construct_derived()
        );

    if (!d) {
       throw std::runtime_error(
```

```
            "Failed to dynamic cast"
        );
    }

    d->derived_method();
    im_accepting_derived(d);
}
```

Как это работает...

В библиотеке Boost имеется множество функций для преобразования умных указателей. Все они принимают умный указатель и параметр шаблона T, где T – требуемый результирующий тип шаблонного параметра указателя. Эти функции преобразования имитируют поведение встроенных операторов приведений, при этом не забывают правильно управлять подсчетом ссылок:

- `boost::static_pointer_cast<T>(p)` → `static_cast<T*>(p.get())`
- `boost::dynamic_pointer_cast<T>(p)` → `dynamic_cast<T*>(p.get())`
- `boost::const_pointer_cast<T>(p)` → `const_cast<T*>(p.get())`
- `boost::reinterpret_pointer_cast<T>(p)` → `reinterpret_cast<T*>(p.get())`

Дополнительно...

Все функции `boost::*_pointer_cast` могут работать с умными указателями из стандартной библиотеки и указателями C, если вы подключите заголовочный файл `<boost/pointer_cast.hpp>`.

С C++11 у стандартной библиотеки имеются `std::static_pointer_cast`, `std::dynamic_pointer_cast` и `std::const_pointer_cast`, определенные в заголовочном файле `<memory>`, однако они работают только с `std::shared_ptr`. В стандартной библиотеке C++17 есть `std::reinterpret_pointer_cast`, но он тоже поддерживает работу лишь с `std::shared_ptr`.

См. также

- В документации библиотеки `Boost.SmartPointer` содержатся дополнительные примеры приведения указателей для стандартной библиотеки: http://boost.org/libs/smart_ptr/pointer_cast.html;
- ссылка на приведение для `boost::shared_ptr` доступна по адресу http://boost.org/libs/smart_ptr/shared_ptr.htm;
- рецепт «*Приведение полиморфных объектов*» этой главы покажет вам более подходящий способ динамического приведения.

Приведение полиморфных объектов

Представьте, что какой-то программист спроектировал такой ужасный интерфейс (это хороший пример того, как не следует писать интерфейсы):

```
struct object {
    virtual ~object() {}
};

struct banana: public object {
    void eat() const {}
```

```
    virtual ~banana(){}
};

struct penguin: public object {
    bool try_to_fly() const {
        return false; // пингвины не летают
    }
    virtual ~penguin(){}
};

object* try_produce_banana();
```

Наша задача – написать функцию, которая вызывает `try_produce_banana()` и выбрасывает исключение, если вместо банана из `try_produce_banana()` пришло что-то другое. `try_produce_banana()` может вернуть `nullptr`, поэтому если мы разыменовываем возвращаемое ею значение без проверки, то подвергаем себя опасности разыменовать нулевой указатель.

Подготовка

Для этого рецепта требуются базовые знания C++.

Как это делается...

Итак, нам нужно написать следующий код:

```
void try_eat_banana_impl1() {
    const object* obj = try_produce_banana();
    if (!obj) {
        throw std::bad_cast();
    }
    dynamic_cast<const banana&>(*obj).eat();
}
```

Выглядит ужасно, не правда ли? Библиотека `Boost.Conversion` предлагает решение получше:

```
#include <boost/cast.hpp>

void try_eat_banana_impl2() {
    const object* obj = try_produce_banana();
    boost::polymorphic_cast<const banana*>(obj)->eat();
}
```

Как это работает...

Функция `boost::polymorphic_cast` просто содержит в себе код из первого примера. Вот и все. Функция проверяет ввод на нулевое значение, а затем пытается выполнить динамическое приведение. Любая ошибка во время этих операций выбросит исключение `std::bad_cast`.

Дополнительно...

У библиотеки `Boost.Conversion` также имеется функция `polymorphic_downcast`. Ее нужно использовать только для нисходящего (от базового класса к наслед-

нику) приведения, которое, согласно логике вашего приложения, обязано быть успешным. В режиме отладки (когда макрос `NDEBUG` не определен) она проверит правильность понижающего приведения с помощью оператора `dynamic_cast`. Когда `NDEBUG` определен, функция `polymorphic_downcast` просто выполняет операцию `static_cast`. Эта функция хорошо подходит для использования в критичных к производительности участках кода. При этом у вас остается обнаружение ошибок в отладочных сборках.

Начиная с Boost версии 1.58 существуют функции `boost::polymorphic_pointer_downcast` и `boost::polymorphic_pointer_cast` в библиотеке `Boost.Conversion`. Эти функции позволяют безопасно приводить умные указатели и имеют те же характеристики, что и `boost::polymorphic_cast` и `boost::polymorphic_downcast`.

В стандартной библиотеке C++ аналоги `polymorphic_cast` и `polymorphic_downcast` отсутствуют.

См. также

- Первоначально идея `polymorphic_cast` была предложена в книге «*Язык программирования C++*» Бьёрна Страуструпа. Обратитесь к этой книге для получения дополнительной информации и просто хороших идей;
- официальная документация также может быть полезной; она доступна на странице http://boost.org/libs/conversion;
- обратитесь к предыдущему рецепту для получения дополнительной информации о приведении умных указателей.

Синтаксический анализ (parsing) простого ввода

Парсинг небольшого текста – обычная задача. Но ее решение всегда является дилеммой: будем мы использовать сторонние профессиональные и проверенные инструменты для синтаксического анализа, такие как Bison или ANTLR, или попытаемся написать все вручную, используя только C++ и стандартную библиотеку? Сторонние инструменты хороши для разбора сложных текстов, и с их помощью легко написать парсеры. Однако они требуют дополнительных программ для генерации C++- или C-кода из грамматики, добавляют зависимостей в проект и усложняют настройку вашей сборочной системы.

Рукописные парсеры, как правило, трудно поддерживать. Однако они не требуют ничего, кроме компилятора C++.

Что делает типичный парсер

Давайте начнем с очень простой задачи парсинга даты в формате ISO:

```
YYYY-MM-DD
```

Ниже приведены примеры возможного ввода:

```
2013-03-01
2012-12-31 // ура, вот-вот наступит новый год!
```

Возьмем грамматику для парсера по следующей ссылке: http://www.ietf.org/rfc/rfc333:

```
date-fullyear     = 4DIGIT
date-month        = 2DIGIT  ; 01-12
date-mday         = 2DIGIT  ; 01-28, 01-29, 01-30, 01-31 в зависимости; от месяца и года
full-date         = date-fullyear "-" date-month "-" date-mday
```

Подготовка

Убедитесь, что вы знакомы с концепцией заполнителя или прочитали рецепт «*Привязка и переупорядочение параметров функции*» главы 1 «Приступаем к написанию приложения». Базовые знания инструментов синтаксического анализа были бы кстати.

Как это делается...

Позвольте представить вам библиотеку Boost.Spirit. Она позволяет писать понятные вашему компилятору синтаксические анализаторы (а также лексеры и генераторы) непосредственно в C++-коде. Формат описания грамматик для Boost.Spirit очень близок к **расширенной форме Бэкуса–Наура (РБНФ, Extended Backus–Naur Form – EBNF). РБНФ** используется для описания грамматик в различных международных стандартах и популярных инструментах синтаксического разбора. Грамматика в начале этой главы дается как раз в формате РБНФ:

1. Нам нужно включить в код следующие заголовочные файлы:

```
#include <boost/spirit/include/qi.hpp>
#include <boost/spirit/include/phoenix_core.hpp>
#include <boost/spirit/include/phoenix_operator.hpp>
#include <cassert>
```

2. Теперь пришло время создать структуру date для хранения данных:

```
struct date {
    unsigned short year;
    unsigned short month;
    unsigned short day;
};
```

3. Давайте посмотрим на парсер (пошаговое описание того, как он работает, можно найти в следующем разделе):

```
// В рецепте "Использование типа «ссылка на строку»" можно найти более подходящий
// тип для параметра 's'
date parse_date_time1(const std::string& s) {
    using boost::spirit::qi::_1;
```

```
    using boost::spirit::qi::ushort_;
    using boost::spirit::qi::char_;
    using boost::phoenix::ref;

    date res;
    const char* first = s.data();
    const char* const end = first + s.size();
    const bool success = boost::spirit::qi::parse(first, end,

        // Реализация правила «date-fullyear» из грамматики РБНФ.
        ushort_[ ref(res.year) = _1 ] >> char_('-')
           >> ushort_[ ref(res.month) = _1 ] >> char_('-')
           >> ushort_[ ref(res.day) = _1 ]
   );
   if (!success || first != end) {
       throw std::logic_error("Parsing failed");
  }
  return res;
}
```

4. Теперь мы можем использовать этот парсер, где угодно:

```
int main() {
    const date d = parse_date_time1("2017-12-31");
    assert(d.year == 2017);
    assert(d.month == 12);
    assert(d.day == 31);
}
```

Как это работает...

Мы получили очень простую реализацию парсера, которая не проверяет количество цифр во входных данных. Синтаксический анализ происходит в функции `boost::spirit::qi::parse`. Давайте немного упростим его реализацию, убрав семантические действия (действия, выполняемые при успешном парсинге):

```
const bool success = boost::spirit::qi::parse(first, end,
     ushort_ >> char_('-') >> ushort_ >> char_('-') >> ushort_
);
```

Аргумент `first` указывает на начало данных для анализа. Это должна быть неконстантная переменная, потому что функция `parse` изменит ее так, чтобы она указывала на конец проанализированной последовательности.

Аргумент `end` указывает на позицию, которая идет после последнего элемента для анализа. `first` и `end` должны быть итераторами или указателями.

Третий аргумент функции – это правило синтаксического анализа. Этот параметр выглядит практически как правило РБНФ:

```
date-fullyear "-" date-month "-" date-md
```

Мы просто заменили пробелы оператором `>>`.

В случае успеха функция `parse` возвращает значение `true`. Если мы хотим убедиться, что вся строка была успешно проанализирована, нам нужно проверить

возвращаемое значение синтаксического анализатора, а также равенство end и модифицированного итератора first.

Осталось разобраться с действиями при успешном разборе, и этот рецепт будет закончен.

Семантические действия в Boost.Spirit пишутся внутри скобок []. Можно использовать указатели на функции, функциональные объекты, boost::bind, std::bind, другие реализации bind() или лямбда-функции C++11.

Вы также можете написать правило для YYYY, используя лямбда-выражение C++11:

```
const auto y = [&res](unsigned short s) { res.year = s; };
// ...
ushort_[y] >> char_('-') >> // ...
```

Нельзя помещать определение лямбда-выражения непосредственно в скобки [], потому что компилятор C++ подумает, что это атрибут. В качестве обходного пути вы можете создать auto переменную с лямбда-функцией в ней и использовать эту переменную в описании правила синтаксического анализатора (так же, как это было сделано в предыдущем фрагменте кода).

Теперь давайте подробнее рассмотрим семантическое действие для разбора месяца:

```
ushort_[ ref(res.month) = _1 ]
```

Тем, кто читает книгу с самого начала, этот код напоминает о boost::bind, boost::ref . ref(res.month) означает передачу res.month в качестве модифицируемой ссылки, а _1 означает первый входной параметр – число, результат парсинга в правиле ushort_.

Дополнительно...

Теперь давайте изменим наш парсер, чтобы он учитывал количество цифр. Для этого мы возьмем шаблон класса unit_parser и просто установим правильные параметры:

```
date parse_date_time2(const std::string& s) {
    using boost::spirit::qi::_1;
    using boost::spirit::qi::uint_parser;
    using boost::spirit::qi::char_;
    using boost::phoenix::ref;
    date res;

    // Используем unsigned short в качестве выходного типа; требуется система счисления
    // по основанию 10 и от 2 до 2 цифр (то есть ровно 2 цифры).
    uint_parser<unsigned short, 10, 2, 2> u2_;

    // Используем unsigned short в качестве выходного типа; требуется десятичная
    // система счисления и от 4 до 4 цифр (то есть ровно 4 цифры).
    uint_parser<unsigned short, 10, 4, 4> u4_;

    const char* first = s.data();
    const char* const end = first + s.size();
```

```
    const bool success = boost::spirit::qi::parse(first, end,

        u4_ [ ref(res.year) = _1 ] >> char_('-')
            >> u2_ [ ref(res.month) = _1 ] >> char_('-')
            >> u2_ [ ref(res.day) = _1 ]

    );
    if (!success || first != end) {
        throw std::logic_error("Parsing failed");
    }
    return res;
}
```

Не волнуйтесь, если эти примеры кажутся сложными. В первый раз я тоже был напуган библиотекой `Boost.Spirit`, но теперь она упрощает мне жизнь. Если этот код не пугает вас, то вы очень смелый человек.

Не пишите парсеры в заголовочных файлах, потому что это увеличивает время компиляции вашего проекта. Пишите их в `.cpp` - исходных файлах и спрячьте там всю работу с `Boost.Spirit`. Если мы подправим предыдущий пример, чтобы следовать этому правилу, тогда заголовочный файл будет выглядеть так:

```
// Заголовочный файл
#ifndef MY_PROJECT_PARSE_DATE_TIME
#define MY_PROJECT_PARSE_DATE_TIME

#include <string>

struct date {
    unsigned short year;
    unsigned short month;
    unsigned short day;
};

date parse_date_time2(const std::string& s);

#endif // MY_PROJECT_PARSE_DATE_TIME
```

Также позаботьтесь о типах итераторов, передаваемых в функцию `boost::spirit::parse`. Чем меньше разных типов итераторов вы используете, тем меньший по размеру бинарный файл вы получаете.

Если вы думаете, что парсинг даты проще было бы реализовать вручную с помощью стандартной библиотеки, то вы правы! Но это только пока. Взгляните на следующий рецепт, в котором написание парсера вручную сложнее, нежели использование `Boost.Spirit`.

Библиотека `Boost.Spirit` не является частью стандарта C++ и не будет предложена для включения в ближайшее время. Однако она довольно хорошо работает с современными возможностями C++, поэтому используйте их, если ваш компилятор поддерживает C++11:

```
date parse_date_time2_cxx(const std::string& s) {
    using boost::spirit::qi::uint_parser;
    using boost::spirit::qi::char_;

    date res;
```

```
    uint_parser<unsigned short, 10, 2, 2> u2_;
    uint_parser<unsigned short, 10, 4, 4> u4_;

    const auto y = [&res](unsigned short s) { res.year = s; };
    const auto m = [&res](unsigned short s) { res.month = s; };
    const auto d = [&res](unsigned short s) { res.day = s; };
    const char* first = s.data();
    const char* const end = first + s.size();
    const bool success = boost::spirit::qi::parse(first, end,
        u4_[y] >> char_('-') >> u2_[m] >> char_('-') >> u2_[d]
    );
    if (!success || first != end) {
        throw std::logic_error("Parsing failed");
    }
    return res;
}
```

См. также

- Рецепт «*Переупорядочение параметров функции*» из главы 1 «*Приступаем к написанию приложения*»;
- рецепт «*Связывание значения как параметра функции*»;
- `Boost.Spirit` – огромная библиотека header-only. О ней можно написать отдельную книгу. Не стесняйтесь использовать документацию по ней по адресу http://boost.org/libs/spirit.

Синтаксический анализ (parsing) сложного ввода

В предыдущем рецепте мы писали простой парсер для даты. Представьте, что прошло какое-то время и задача изменилась. Теперь нам нужно написать парсер даты и времени, который поддерживает несколько форматов ввода и часовые пояса. Наш парсер должен понимать следующие входные данные:

```
2012-10-20T10:00:00Z      // Дата и время с нулевым смещением поясов
2012-10-20T10:00:00       // Дата и время без смещения поясов
2012-10-20T10:00:00+09:15 // Дата и время со смещением поясов
2012-10-20-09:15          // Дата и время без смещения поясов
10:00:09+09:15            // Время со смещением поясов
```

Подготовка

Мы будем использовать библиотеку `Boost.Spirit`, которая была описана в рецепте «*Синтаксический анализ (parsing) простого ввода*». Прочтите его, прежде чем приступить к этому рецепту.

Как это делается...

1. Давайте начнем с написания структуры даты и времени, которая будет содержать результат:

```
#include <stdexcept>
#include <cassert>
```

```
struct datetime {
    enum zone_offsets_t {
        OFFSET_NOT_SET,
        OFFSET_Z,
        OFFSET_UTC_PLUS,
        OFFSET_UTC_MINUS
    };

private:
    unsigned short year_;
    unsigned short month_;
    unsigned short day_;

    unsigned short hours_;
    unsigned short minutes_;
    unsigned short seconds_;

    zone_offsets_t zone_offset_type_;
    unsigned int zone_offset_in_min_;

    static void dt_assert(bool v, const char* msg) {
        if (!v) {
            throw std::logic_error(
                "Assertion failed in datetime: " + std::string(msg)
            );
        }
    }

public:
    datetime()
        : year_(0), month_(0), day_(0)
        , hours_(0), minutes_(0), seconds_(0)
        , zone_offset_type_(OFFSET_NOT_SET), zone_offset_in_min_(0)
    {}

    // Геттеры: year(), month(), day(), hours(), minutes(),
    // seconds(), zone_offset_type(), zone_offset_in_min()
    // ...
    // Сеттеры: set_year(unsigned short), set_day(unsigned short), ...
    //
    // void set_*(unsigned short val) {
        // Some dt_assert.
        // Устанавливаем для '*_' значение 'val'.
   // }
   // ...
};
```

2. Теперь давайте напишем функцию для выставления смещения поясов:

```
void set_zone_offset(datetime& dt, char sign, unsigned short hours
    , unsigned short minutes)
{
```

```
    dt.set_zone_offset(
        sign == '+'
        ? datetime::OFFSET_UTC_PLUS
        : datetime::OFFSET_UTC_MINUS
    );
    dt.set_zone_offset_in_min(hours * 60 + minutes);
}
```

3. Наш парсер можно собрать из нескольких простых парсеров. Начнем с написания парсера смещения поясов:

```
// Основные заголовочные файлы для Boost.Spirit.
#include <boost/spirit/include/qi.hpp>
#include <boost/spirit/include/phoenix_core.hpp>
#include <boost/spirit/include/phoenix_operator.hpp>

// Мы будем использовать функцию bind() из Boost.Spirit, потому что она лучше
// взаимодействует с парсерами.
#include <boost/spirit/include/phoenix_bind.hpp>

datetime parse_datetime(const std::string& s) {
    using boost::spirit::qi::_1;
    using boost::spirit::qi::_2;
    using boost::spirit::qi::_3;
    using boost::spirit::qi::uint_parser;
    using boost::spirit::qi::char_;
    using boost::phoenix::bind;
    using boost::phoenix::ref;

    datetime ret;

   // Используем unsigned short в качестве выходного типа; требуется
   // десятичная система счисления и от 2 до 2 цифр (то есть ровно 2 цифры).
   uint_parser<unsigned short, 10, 2, 2> u2_;

   // Используем unsigned short в качестве выходного типа; требуется
   // десятичная система счисления и от 4 до 4 цифр (то есть ровно 4 цифры).
   uint_parser<unsigned short, 10, 4, 4> u4_;

   boost::spirit::qi::rule<const char*, void()> timezone_parser
       = -( // унарный минус значит “опциональное” правило
            // Нулевое смещение
            char_('Z')[ bind(
                 &datetime::set_zone_offset, &ret,
datetime::OFFSET_Z
            ) ]

            | // ИЛИ

            // Смещение задано цифрами
            ((char_('+')|char_('-')) >> u2_ >> ':' >> u2_) [
                bind(&set_zone_offset, ref(ret), _1, _2, _3)
            ]
      );
```

4. Давайте закончим наш пример, написав оставшиеся парсеры и объединив их в один:

```
boost::spirit::qi::rule<const char*, void()> date_parser =
       u4_ [ bind(&datetime::set_year, &ret, _1) ] >> '-'
    >> u2_ [ bind(&datetime::set_month, &ret, _1) ] >> '-'
    >> u2_ [ bind(&datetime::set_day, &ret, _1) ];

boost::spirit::qi::rule<const char*, void()> time_parser =
        u2_ [ bind(&datetime::set_hours, &ret, _1) ] >> ':'
     >> u2_ [ bind(&datetime::set_minutes, &ret, _1) ] >> ':'
     >> u2_ [ bind(&datetime::set_seconds, &ret, _1) ];

const char* first = s.data();
const char* const end = first + s.size();
const bool success = boost::spirit::qi::parse(first, end,
    (
        (date_parser >> 'T' >> time_parser)
        | date_parser
        | time_parser
    )
    >> timezone_parser
);

if (!success || first != end) {
    throw std::logic_error("Parsing of '" + s + "' failed");
}
return ret;
} // Конец функции parse_datetime()
```

Как это работает....

Здесь есть очень интересная переменная с типом `boost::spirit::qi::rule<const char *, void()>`. При сохранении парсера в такую переменную его точный тип будет стерт. Это позволяет писать парсеры для рекурсивных грамматик, а также писать парсеры в .cpp-исходных файлах и использовать их в заголовочных файлах без существенного влияния на время компиляции вашего проекта.

Например:

```
// Где-то в заголовочном файле
class example_1 {
   boost::spirit::qi::rule<const char*, void()> some_rule_;
public:
    example_1();
};

// В cpp-исходном файле
example_1::example_1() {
    some_rule_ = /* ... Большое количество кода парсера ... */;
}
```

Обратите внимание, что этот класс является барьером оптимизации для компиляторов, поэтому не используйте его, когда это не требуется.

В примерах иногда мы использовали >> ':' вместо >> char_(':'). Первый подход более ограничен: вы не можете привязать к нему семантические действия и не можете создавать новые правила, сочетая символы (например, нельзя реализовать парсер наподобие char_('+')|char_('-'), вообще не используя char_).

Но для лучшей производительности используйте первый подход, потому что некоторые компиляторы могут оптимизировать его чуточку лучше.

Дополнительно...

Мы можем сделать наш пример немного быстрее, удалив объекты rule<>, которые стирают типы. Просто замените их типы на ключевое слово auto из C++11 и вызовите boost::proto::deep_copy() для грамматики:

```
const auto date_parser = boost::proto::deep_copy(
       u4_ [ bind(&datetime::set_year, &ret, _1) ]  >> '-'
    >> u2_ [ bind(&datetime::set_month, &ret, _1) ] >> '-'
    >> u2_ [ bind(&datetime::set_day, &ret, _1) ]
);
```

Библиотека Boost.Spirit генерирует очень быстрые парсеры; на официальном сайте есть показатели эффективности. Документация содержит расширенные рекомендации по написанию более быстрых парсеров.

Использование функции boost::phoenix::bind не обязательно, но без нее правило, которое анализирует конкретное смещение поясов в timezone_parser, будет иметь дело с типом boost::fusion::vector<char, unsigned short, unsigned short>. Использование bind(& set_zone_offset, ref (ret), _1, _2, _3) представляется более удобным для читателя решением.

При синтаксическом анализе больших файлов прочтите рецепт «*Самый быстрый способ чтения файлов*» главы 11 «Работа с системой», поскольку некорректная работа с файлами может замедлить работу вашей программы гораздо сильнее, чем синтаксический анализ.

Компиляция кода, использующего библиотеку Boost.Spirit (или Boost.Fusion), может занять много времени из-за огромного количества инстанцирований шаблонов. При экспериментах с библиотекой Boost.Spirit рекомендуется использовать современные компиляторы, они обеспечивают лучшее время компиляции.

См. также

Про библиотеку Boost.Spirit стоит написать отдельную книгу. Невозможно описать ее в нескольких рецептах, поэтому если вы обратитесь к документации, это поможет вам получить больше информации о ней. Документация доступна по адресу: http://boost.org/libs/spirit.

Глава 4

Уловки времени компиляции

Темы, которые мы рассмотрим в этой главе:

- проверка размеров во время компиляции;
- активация использования шаблона функции для целочисленных типов;
- отключение использования шаблона функции для действительных типов;
- создание типа из числа;
- реализация собственного type trait;
- выбор оптимального оператора для шаблонного параметра;
- получение типа выражения в C++03.

Вступление

В этой главе мы увидим несколько базовых примеров того, как можно использовать библиотеки Boost при проверках времени компиляции, для настройки алгоритмов и в других задачах метапрограммирования.

Некоторые читатели могут спросить: «Зачем нам нужны все эти вещи, связанные со временем компиляции?» Дело в том, что когда вы отправляете программу заказчику, вы ее компилируете один раз, а заказчик запускает вашу программу множество раз. Чем больше мы делаем во время компиляции, тем меньше времени и системных ресурсов тратится при работе вашего приложения, что приводит к появлению гораздо более быстрых и надежных программ. Проверки во время выполнения (runtime) отработают, только если выполнится часть кода с проверкой. Проверка времени компиляции прервет компиляцию вашей программы, в идеале вы увидите осмысленное сообщение об ошибке от компилятора.

Эта глава, пожалуй, одна из самых важных. Понять исходники Boost и другие похожие на Boost библиотеки без нее невозможно.

Проверка размеров во время компиляции

Давайте представим, что мы пишем некую функцию сериализации, которая хранит значения в буфере указанного размера:

```
#include <cstring>
#include <boost/array.hpp>

// В C++17 есть готовый тип std::byte!
// К сожалению, это пример C++03.
typedef unsigned char byte_t;
```

```
template <class T, std::size_t BufSizeV>
void serialize_bad(const T& value, boost::array<byte_t, BufSizeV>& buffer)
{
    // TODO: проверить размер буфера.
    std::memcpy(&buffer[0], &value, sizeof(value));
}
```

Этот код имеет следующие проблемы:

- размер буфера не проверяется, поэтому буфер может переполниться;
- эту функцию можно использовать с нетривиально копируемыми типами, что приведет к некорректному поведению.

Мы можем частично исправить эти проблемы, добавив assert, например:

```
template <class T, std::size_t BufSizeV>
void serialize_bad(const T& value, boost::array<byte_t, BufSizeV>& buffer)
{
    // TODO: придумать что-нибудь получше.
    assert(BufSizeV >= sizeof(value));
    std::memcpy(&buffer[0], &value, sizeof(value));
}
```

Но это плохое решение. Проверка времени выполнения `assert` не отработает в режиме отладки, если функция не была вызвана. Проверки времени выполнения в режиме выпуска (release), как правило, выкидываются компилятором, поэтому вы не заметите проблему.

При этом значения `BufSizeV` и `sizeof(value)` известны на этапе компиляции. Это означает, что вместо проверки времени выполнения мы можем просто прервать компиляцию кода, если буфер слишком мал.

Подготовка

Этот рецепт требует знания шаблонов C++ и библиотеки `Boost.Array`.

Как это делается...

Давайте воспользуемся библиотеками `Boost.StaticAssert` и `Boost.TypeTraits`:

```
#include <boost/static_assert.hpp>
#include <boost/type_traits/has_trivial_copy.hpp>

template <class T, std::size_t BufSizeV>
void serialize(const T& value, boost::array<byte_t, BufSizeV>& buffer) {
    BOOST_STATIC_ASSERT(BufSizeV >= sizeof(value));
    BOOST_STATIC_ASSERT(boost::has_trivial_copy<T>::value);

    std::memcpy(&buffer[0], &value, sizeof(value));
}
```

Как это работает...

Макрос `BOOST_STATIC_ASSERT` можно использовать только в том случае, если выражение для проверки вычислимо во время компиляции и его результат можно не-

явно преобразовать к типу bool. Это означает, что вы можете использовать функцию sizeof(), статические константы, constexpr-переменные, constexpr-функции с параметрами, известными во время компиляции, и другие константные выражения в BOOST_STATIC_ASSERT. Если результат вычисления выражения имеет значение false, BOOST_STATIC_ASSERT остановит компиляцию. В случае функции serialize, если первая статическая проверка не выполняется, это означает, что пользователь неправильно использовал функцию и предоставил очень маленький буфер.

Вот еще несколько примеров:

```
BOOST_STATIC_ASSERT(3 >= 1);

struct some_struct { enum enum_t { value = 1}; };
BOOST_STATIC_ASSERT(some_struct::value);

template <class T1, class T2>
struct some_templated_struct
{
    enum enum_t { value = (sizeof(T1) == sizeof(T2))};
};
BOOST_STATIC_ASSERT((some_templated_struct<int, unsigned int>::value));

template <class T1, class T2>
struct some_template {
    BOOST_STATIC_ASSERT(sizeof(T1) == sizeof(T2));
};
```

Если выражение внутри макроса BOOST_STATIC_ASSERT содержит знак запятой, мы должны заключить все выражение в дополнительные круглые скобки.

Итак, теперь пришло время узнать больше о библиотеке Boost.TypeTraits. Эта библиотека предоставляет большое количество метафункций, которые позволяют нам на этапе компиляции получать информацию о типах и изменять их. Использование метафункций выглядит как boost::function_name<parameters>::value или boost::function_name<parameters>::type. Метафункция boost::has_trivial_copy<T>::value возвращает значение true, только если T – это простой копируемый тип.

Давайте посмотрим еще на несколько примеров:

```
#include <iostream>
#include <boost/type_traits/is_unsigned.hpp>
#include <boost/type_traits/is_same.hpp>
#include <boost/type_traits/remove_const.hpp>

template <class T1, class T2>
void type_traits_examples(T1& /*v1*/, T2& /*v2*/) {
     // Возвращает true, если T1 - беззнаковый тип числа
     std::cout << boost::is_unsigned<T1>::value;

     // Возвращает true, если T1 имеет точно такой же тип, как и T2
     std::cout << boost::is_same<T1, T2>::value;

     // Эта строка удаляет модификатор const из типа T1.
```

```
    // Вот как будет меняться тип T1 под
    // действием boost::remove_const:
    // const int => int
    // int => int
    // int const volatile => int volatile
    // const int& => const int&
    typedef typename boost::remove_const<T1>::type t1_nonconst_t;
}
```

Некоторые компиляторы могут компилировать этот код даже без ключевого слова `typename`, но такое поведение нарушает стандарт C++, поэтому рекомендуется использовать `typename`.

Дополнительно...

У макроса `BOOST_STATIC_ASSSERT` есть более многословный вариант, `BOOST_STATIC_ASSSERT_MSG`, который изо всех сил пытается вывести сообщение об ошибке в журнале (логах, logs) компилятора или в окне интегрированной среды разработки. Посмотрите на приведенный ниже код:

```
template <class T, std::size_t BufSizeV>
void serialize2(const T& value, boost::array<byte_t, BufSizeV>& buf) {
    BOOST_STATIC_ASSERT_MSG(boost::has_trivial_copy<T>::value,
        "This serialize2 function may be used only "
        "with trivially copyable types."
    );

    BOOST_STATIC_ASSERT_MSG(BufSizeV >= sizeof(value),
        "Can not fit value to buffer. "
        "Make the buffer bigger."
    );

    std::memcpy(&buf[0], &value, sizeof(value));
}

int main() {
    // Где-то в коде:
    boost::array<unsigned char, 1> buf;
    serialize2(std::string("Hello word"), buf);
}
```

Предыдущий код во время компиляции на g++-компиляторе в режиме C++11 выведет следующее сообщение в консоль:

```
boost/static_assert.hpp:31:45: error: static assertion failed: This serialize2 function
may be used only with trivially copyable types.
 # define BOOST_STATIC_ASSERT_MSG( ... ) static_assert(__VA_ARGS__)
                                        ^
Chapter04/01_static_assert/main.cpp:76:5: note: in expansion of macro ‘BOOST_STATIC_
ASSERT_MSG;
    BOOST_STATIC_ASSERT_MSG(boost::has_trivial_copy<T>::value,
    ^~~~~~~~~~~~~~~~~~~~~~~
```

```
boost/static_assert.hpp:31:45: error: static assertion failed: Can not fit value to
buffer. Make the buffer bigger.
 # define BOOST_STATIC_ASSERT_MSG( ... ) static_assert(__VA_ARGS__)
                                         ^

Chapter04/01_static_assert/main.cpp:81:5: note: in expansion of macro ‘BOOST_STATIC_
ASSERT_MSG;
    BOOST_STATIC_ASSERT_MSG(BufSizeV >= sizeof(value),
           ^~~~~~~~~~~~~~~~~~~~~~
```

Ни BOOST_STATIC_ASSSERT, ни BOOST_STATIC_ASSSERT_MSG, ни какие-либо из type traits не добавляют накладных расходов времени выполнения. Все эти функции выполняются только во время компиляции и не добавляют ни одной машинной инструкции в итоговый двоичный файл. В стандарте C++11 существует static_assert(condition, "message"), эквивалентный макросу BOOST_STATI_ASSSERT_MSG.

Функциональность BOOST_STATIC_ASSERT проверок времени компиляции без пользовательских сообщений доступна в C++17 в виде static_assert(condition). Вам не нужно подключать заголовочные файлы, чтобы иметь возможность использовать встроенный в ваш компилятор static_assert.

Библиотека Boost.TypeTraits была частично принята в стандарт C++11. Таким образом, вы можете найти нужные вам метафункции в заголовке <type_traits> в пространстве имен std::. Метафункции, имена которых в Boost начинаются с has_, переименованы в стандартной библиотеке в метафункции с именами, начинающимися с is_.

Таким образом, boost::has_trivial_copy стал std::is_trivially_copyable и т. д.

C++14 и Boost с версии 1.65 имеют псевдонимы типов (type alias) для всех метафункций, у которых есть член ::type. Эти псевдонимы позволяют писать remove_const_t<T1> вместо typename remove_const<T1>::type. Обратите внимание, что в случае с Boost версии 1.65 псевдонимам требуется компилятор, совместимый с C++11, из-за механизма их реализации:

```
template <class T>
using remove_const_t = typename remove_const<T>::type;
```

В C++17 добавлены шаблонные переменные, заканчивающиеся на _v, для значений метафункций ::value. Начиная с C++17 можно просто написать std::is_unsigned_v<T1> вместо std::is_unsigned<T1>::value. Реализовываются подобные шаблонные переменные обычно вот так:

```
template <class T>
inline constexpr bool is_unsigned_v = is_unsigned<T>::value;
```

Если в Boost и в стандартной библиотеке C++ есть похожая метафункция, выбирайте Boost-версию, если ваш проект должен собираться с версиями стандарта C++, вышедшими до 2011 года.

Иначе стандартная библиотека будет работать немного лучше на краевых случаях.

См. также

- Следующие рецепты из этой главы предоставят вам больше примеров и идей того, как можно использовать статические проверки и метафункции;

- прочитайте официальную документацию по библиотеке Boost.StaticAssert, чтобы найти еще больше примеров: http://boost.org/libs/static_assert.

Активация использования шаблона функции для интегральных типов

У нас есть шаблон класса, реализующий некую функциональность:

```
// Универсальная реализация.
template <class T>
class data_processor {
    double process(const T& v1, const T& v2, const T& v3);
};
```

Теперь представьте, что у нас есть две дополнительные реализации этого класса, одна для интегральных, а другая для действительных типов:

```
// Оптимизированная версия для интегральных типов
template <class T>
class data_processor_integral {
    typedef int fast_int_t;
    double process(fast_int_t v1, fast_int_t v2, fast_int_t v3);
};

// Оптимизированная SSE-версия для типов с плавающей точкой
template <class T>
class data_processor_sse {
    double process(double v1, double v2, double v3);
};
```

Теперь вопрос: как заставить компилятор автоматически выбирать оптимальную реализацию в зависимости от шаблонного параметра?

Подготовка

Этот рецепт требует базовых знаний шаблонов C++.

Как это делается...

Мы будем использовать библиотеки Boost.Core и Boost.TypeTraits для решения.

1. Давайте начнем с подключения заголовочных файлов:

```
#include <boost/core/enable_if.hpp>
#include <boost/type_traits/is_integral.hpp>
#include <boost/type_traits/is_float.hpp>
```

2. Добавим дополнительный шаблонный параметр со значением по умолчанию к нашей универсальной реализации:

```
// Универсальная реализация
template <class T, class Enable = void>
class data_processor {
    // ...
};
```

3. Изменим оптимизированные версии, превратив их в частичные специализации шаблона:

```
// Оптимизированная версия для интегральных типов
template <class T>
class data_processor<
    T,
    typename boost::enable_if_c<boost::is_integral<T>::value
>::type
>
{
    // ...
};

// Оптимизированная SSE-версия для типов с плавающей точкой
template <class T>
class data_processor<
    T,
    typename boost::enable_if_c<boost::is_float<T>::value >::type
>{
    // ...
};
```

4. Вот и все! Теперь компилятор автоматически выберет правильный класс:

```
template <class T>
double example_func(T v1, T v2, T v3) {
    data_processor<T> proc;
    return proc.process(v1, v2, v3);
}

int main () {
     // Будет вызвана оптимизированная версия для интегральных типов.
     example_func(1, 2, 3);
     short s = 0;
     example_func(s, s, s);

     // Будет вызвана версия для действительных типов
     example_func(1.0, 2.0, 3.0);
     example_func(1.0f, 2.0f, 3.0f);

     // Будет вызвана универсальная версия
     example_func("Hello", "word", "processing");
}
```

Как это работает...

Шаблон `boost::enable_if_c` весьма заковыристый. В нем используется принцип **SFINAE** (англ. **substitution failure is not an error**, «неудавшаяся подстановка – не ошибка»), который применяется компилятором во время **инстанцирования шаблонов**. Вот как работает этот принцип: если невалидное выражение или невалидный возвращаемый тип формируется во время инстанцирования шаблона, инстанцирование «выкидывается» и не вызывает ошибку компиляции.

Теперь о заковырке: у boost::enable_if_c<true> имеется псевдоним типа, доступный через ::type, но у boost::enable_if_c<false> его нет. Давайте вернемся к нашему решению и посмотрим, как принцип SFINAE работает с различными типами, передаваемыми в класс data_processor в качестве параметра T.

Если мы передаем int как тип T, сначала компилятор попытается проинстанцировать частичные специализации шаблона из шага 3, прежде чем использовать нашу неспециализированную универсальную версию из шага 2.

Когда он попытается инстанцировать версию класса для типа с плавающей точкой, метафункция boost::is_float<T>::value вернет значение false. Нельзя проинстанцировать метафункцию boost::enable_if_c<false>::type, потому что boost::enable_if_c<false> не имеет ::type, и вот тут начинает действовать принцип SFINAE. Поскольку шаблон класса невалидный, компилятор игнорирует эту специализацию шаблона. Следующая частичная специализация – та, что для интегральных типов. Метафункция boost::is_integral<T>::value возвращает значение true, и можно проинстанцировать boost::enable_if_c<true>::type, что позволяет проинстанцировать всю специализацию data_processor. Компилятор нашел подходящую частичную специализацию, поэтому не нужно пытаться инстанцировать неспециализованный класс.

Теперь давайте попробуем передать неарифметический тип (например, const char *) и посмотрим, что будет делать компилятор. Сначала компилятор будет пытаться проинстанцировать частичные специализации шаблона. Специализации с is_float<T>::value и is_integral<T>::value нельзя инстанцировать, поэтому компилятор их проигнорирует, будет пытаться проинстанцировать нашу универсальную версию, и ему это удастся.

Без boost::enable_if_c<> все частичные специализации можно проинстанцировать для любого типа, что приводит к неоднозначности и ошибке компиляции.

Если вы используете шаблоны и компилятор вам говорит, что не может выбрать между двумя шаблонными классами, то, вероятно, вам нужен boost::enable_if_c<>.

Дополнительно...

Еще одна версия этого метода называется boost::enable_if без _c в конце. Разница между ними заключается в том, что enable_if_c принимает константу в качестве параметра шаблона; сокращенная версия принимает объект, у которого есть статический член value. Например:

```
boost::enable_if_c<boost::is_integral<T>::value>::type эквивалентно
boost::enable_if<boost::is_integral<T> >::type.
```

До появления Boost версии 1.56 метафункции boost::enable_if определялись в заголовочном файле <boost/utility/enable_if.hpp> вместо <boost/core/enable_if.hpp>.

В C++11 есть std::enable_if, определенный в заголовке <type_traits>, который ведет себя точно так же, как boost::enable_if_c. Никакой разницы между ними нет, за исключением того, что версия для Boost работает и в компиляторах, не поддерживающих C++11, обеспечивая лучшую переносимость.

В C++14 есть псевдоним шаблона `std::enable_if_t`, который должен использоваться без `typename` и `::type`:

```
template <class T>
class data_processor<
    T, std::enable_if_t<boost::is_float<T>::value >
>;
```

Все функции `enable_if` выполняются только во время компиляции и не добавляют накладных расходов времени выполнения. Тем не менее добавление дополнительного параметра шаблона может привести к увеличению имени класса в `typeid(ваш_класс).name()` и добавить очень незначительную нагрузку на процессор при сравнении двух результатов `typeid()` на некоторых платформах.

См. также

- Приведенные далее рецепты дадут вам больше примеров использования `enable_if`;
- вы также можете ознакомиться с официальной документацией по библиотеке `Boost.Core`. В ней содержится много примеров и полезных классов (которые широко используются в этой книге). Перейдите по ссылке http://boost.org/libs/core;
- вы также можете прочитать статьи о частичной специализации шаблона на странице http://msdn.microsoft.com/en-us/library/3967w96f%28v=vs.110%29.aspx.

Отключение использования шаблона функции для действительных типов

Продолжаем работать с библиотеками метапрограммирования Boost. В предыдущем рецепте мы увидели, как использовать `enable_if_c` с классами; теперь пришло время взглянуть, как использовать его в шаблонных функциях.

Представьте, что в вашем проекте есть шаблонная функция, которая работает со всеми доступными типами:

```
template <class T>
T process_data(const T& v1, const T& v2, const T& v3);
```

Эта функция существует в вашем проекте в течение длительного времени. Вы написали много кода, в котором она используется.

Неожиданно вы придумали оптимизированную версию функции `process_data`, но только для типов, которые имеют `T::operator+=(const T&)`:

```
template <class T>
T process_data_plus_assign(const T& v1, const T& v2, const T& v3);
```

У вас огромная кодовая база, и могут потребоваться месяцы, чтобы вручную заменить `process_data` на `process_data_plus_assign` для типов, у которых есть правильные операторы. Итак, вы не хотите менять уже написанный код. Вместо этого вы хотите заставить компилятор автоматически использовать оптимизированную функцию вместо функции по умолчанию, если это возможно.

Подготовка

Прочитайте предыдущий рецепт, чтобы понять, что делает `boost::enable_if_c`, и понять концепцию SFINAE. По-прежнему требуется базовое знание шаблонов.

Как это делается...

Шаблонная магия с использованием библиотек Boost:

1. Нам понадобится метафункция boost::has_plus_assign<T> и заголовок <boost/enable_if.hpp>:

```
#include <boost/core/enable_if.hpp>
#include <boost/type_traits/has_plus_assign.hpp>
```

2. Теперь мы отключаем реализацию по умолчанию для типов с помощью `boost::has_plus_assign`:

```
// Модифицированная универсальная версия функции process_data
template <class T>
typename boost::disable_if_c<boost::has_plus_assign<T>::value, T>::type
    process_data(const T& v1, const T& v2, const T& v3);
```

3. Активируйте оптимизированную версию для типов с помощью `boost::has_plus_assign`:

```
// process_data вызовет process_data_plus_assign
template <class T>
typename boost::enable_if_c<boost::has_plus_assign<T>::value,
T>::type
    process_data(const T& v1, const T& v2, const T& v3)
{
    return process_data_plus_assign(v1, v2, v3);
}
```

4. Теперь оптимизированная версия используется везде, где это возможно:

```
int main() {
    int i = 1;
    // Оптимизированная версия
    process_data(i, i, i);

    // Версия по умолчанию
    // Явное указание параметра шаблона по-прежнему работает.
    process_data<const char*>("Testing", "example", "function");
}
```

Как это работает...

Метафункция `boost::disable_if_c<bool_value>::type` отключает метод, если `bool_value` равно `true`. Она работает так же, как `boost::enable_if_c<!bool_value>::type`.

Класс, переданный в качестве второго шаблонного параметра для `boost::enable_if_c` или `boost::disable_if_c`, возвращается с помощью `::type` в случае успеха. Другими словами, `boost::enable_if_c<true, T>::type` – то же самое, что и `T`.

Давайте шаг за шагом рассмотрим вариант с `process_data(i, i, i)`. Мы передаем `int` как тип `T`, и компилятор ищет функцию `process_data(int, int, int)`. Посколь-

ку такой функции нет, следующим шагом является инстанцирование шаблона process_data. Однако есть две шаблонные функции process_data. Допустим, компилятор начинает инстанцировать шаблон нашей второй (оптимизированной) версии; в этом случае он успешно вычисляет typename boost::enable_if_c<boost::has_plus_assign<T>::value, T>::type и получает возвращаемый тип T. Но компилятор не останавливается; он продолжает попытки инстанцирования и пытается проинстанцировать нашу первую версию функции. Во время вычисления typename boost::disable_if_c<boost::has_plus_assign<T>::value происходит сбой, который не рассматривается как ошибка из-за правила SFINAE. Больше нет шаблонных функций process_data, поэтому компилятор прекращает инстанцирование. Как видите, без enable_if_c и disable_if_c компилятор может проинстанцировать экземпляры обоих шаблонов, возникнет неоднозначность и ошибка компиляции.

Дополнительно...

Как и в случае с enable_if_c и enable_if, существует функция disable_if:

```
// Первая версия
template <class T>
typename boost::disable_if<boost::has_plus_assign<T>, T>::type
    process_data2(const T& v1, const T& v2, const T& v3);

// process_data_plus_assign
template <class T>
typename boost::enable_if<boost::has_plus_assign<T>, T>::type
    process_data2(const T& v1, const T& v2, const T& v3);
```

В C++11 нет ни disable_if_c, ни disable_if, но вы можете спокойно использовать std::enable_if<!bool_value>::type.

До появления Boost версии 1.56 метафункции boost::disable_if определялись в заголовке <boost/utility/enable_if.hpp> вместо <boost/core/enable_if.hpp>.

Как было упомянуто в предыдущем рецепте, все функции enable_if и disable_if выполняются только во время компиляции и не добавляют накладных расходов времени выполнения.

См. также

- Прочтите эту главу с самого начала, чтобы увидеть больше хитростей времени компиляции;
- вы можете прочитать официальную документацию по библиотеке Boost.TypeTraits для получения дополнительных примеров и полного списка метафункций на странице http://boost.org/libs/type_traits;
- библиотека Boost.Core может предоставить вам дополнительные примеры использования boost::enable_if – http://boost.org/libs/core.

Создание типа из числа

Мы увидели примеры того, как можно выбирать функции, используя boost::enable_if_c. Давайте теперь применим другой подход. Рассмотрим сле-

дующий пример, где у нас есть универсальный метод для обработки простых типов данных (POD):

```
#include <boost/static_assert.hpp>
#include <boost/type_traits/is_pod.hpp>

// Универсальная реализация.
template <class T>
T process(const T& val) {
    BOOST_STATIC_ASSERT((boost::is_pod<T>::value));
    // ...
}
```

У нас также есть функции обработки, оптимизированные под размеры 1, 4 и 8 байт. Как переписать функцию `process`, чтобы она могла вызывать оптимизированные функции обработки?

Подготовка

Настоятельно рекомендуется прочитать хотя бы первый рецепт из этой главы, чтобы вас не смутило все происходящее здесь. Шаблоны и метапрограммирования не должны вас пугать.

Как это делается...

Мы рассмотрим, как можно преобразовать размер шаблонного параметра в переменную некоего типа и как можно использовать эту переменную для определения правильной перегрузки функции.

1. Давайте определим универсальную и оптимизированную версии нашей функции `process_impl`:

```
#include <boost/mpl/int.hpp>

namespace detail {
    // Универсальная реализация
    template <class T, class Tag>
    T process_impl(const T& val, Tag /*ignore*/) {
        // ...
    }

    // 1-байтовая оптимизированная реализация.
    template <class T>
    T process_impl(const T& val, boost::mpl::int_<1> /*ignore*/) {
        // ...
    }

    // 4-байтовая оптимизированная реализация.
    template <class T>
    T process_impl(const T& val, boost::mpl::int_<4> /*ignore*/) {
       // ...
    }
```

```
    // 8-байтовая оптимизированная реализация.
    template <class T>
    T process_impl(const T& val, boost::mpl::int_<8> /*ignore*/) {
        // ...
    }
} // пространство имен detail
```

2. Теперь мы готовы написать нашу функцию:

```
// Совершаем вызов:
template <class T>
T process(const T& val) {
    BOOST_STATIC_ASSERT((boost::is_pod<T>::value));
    return detail::process_impl(val, boost::mpl::int_<sizeof(T)>());
}
```

Как это работает...

Самая интересная часть здесь – `boost::mpl::int_<sizeof(T)>()`. `sizeof(T)` выполняется во время компиляции, поэтому его результат может использоваться как параметр шаблона. `boost::mpl::int_<>` – просто пустой класс, который содержит значение времени компиляции для `int`. В библиотеке `Boost.MPL` такие классы называются **интегральными константами**. Его можно реализовать, как показано ниже:

```
template <int Value>
struct int_ {
    static const int value = Value;
    typedef int_<Value> type;
    typedef int value_type;
};
```

Нам нужен экземпляр этого класса, поэтому у нас есть круглые скобки в конце `boost::mpl::int_<sizeof(T)>()`, чтобы вызвать конструктор.

Теперь давайте подробнее рассмотрим, как компилятор выберет функцию `process_impl`. Прежде всего компилятор пытается сопоставить функции, у которых второй параметр не является параметром шаблона. Если `sizeof(T)` равен 4, компилятор пытается найти функцию с такими сигнатурами, как `process_impl(T, boost::mpl::int_<4>)`, и находит нашу 4-байтовую оптимизированную версию из пространства имен `detail`. Если `sizeof(T)` равен 34, компилятор не сможет найти функцию с такой сигнатурой, как `process_impl(T, boost::mpl::int_ <34>)`, и использует шаблонную функцию `process_impl(const T& val, Tag /*ignore*/)`.

Дополнительно...

В библиотеке `Boost.MPL` существует множество структур данных для метапрограммирования. Вам могут пригодиться следующие классы интегральных констант из MPL:

- `bool_`
- `int_`
- `long_`
- `size_t`
- `char_`

Все функции библиотеки Boost.MPL (кроме функции времени выполнения for_each) выполняются во время компиляции и не добавляют накладных расходов времени выполнения.

Библиотека Boost.MPL не является частью стандарта C++. Тем не менее C++ использует множество трюков из этой библиотеки. Начиная с C++11 в заголовочном файле type_traits есть класс std::integral_constant<type, value>, который можно использовать так же, как и в предыдущем примере. Вы даже можете определить собственные псевдонимы типов с его помощью:

```
template <int Value>
using int_ = std::integral_constant<int, Value>;
```

См. также

- В рецептах главы 8 «*Метапрограммирование*» приводятся дополнительные примеры использования библиотеки Boost.MPL. Если вы чувствуете себя уверенно, можете также попробовать прочитать документацию Boost.MPL по ссылке http://boost.org/libs/mpl;
- дополнительные примеры использования тегов можно найти по ссылкам http://boost.org/libs/type_traits/doc/html/boost_typetraits/examples/fill.html и http://boost.org/libs/type_traits/doc/html/boost_typetraits/examples/copy.html.

Реализация свойства типов (type trait)

Нам нужно реализовать свой type trait, который возвращает значение true, если ему в качестве параметра шаблона передается тип std::vector, а в противном случае – false.

Подготовка

Требуются базовые знания о свойствах типов стандартной библиотеки или библиотеки Boost.TypeTrait.

Как это делается...

Давайте посмотрим, как реализовать свойство типов:

```
#include <vector>
#include <boost/type_traits/integral_constant.hpp>

template <class T>
struct is_stdvector: boost::false_type {};

template <class T, class Allocator>
struct is_stdvector<std::vector<T, Allocator> >: boost::true_type {};
```

Как это работает...

Почти всю работу делают классы boost::true_type и boost::false_type. Класс boost::true_type содержит статическую константу ::value, имеющую значение true. Аналогично класс boost::false_type содержит статическую константу ::value, которая имеет значение false.

Эти два класса также имеют typedef, который помогает взаимодействовать с библиотекой Boost.MPL.

Наша первая структура is_stdvector – универсальная, она будет использоваться всегда, когда специализированная шаблонная версия структуры не найдена. Наша вторая структура is_stdvector является шаблонной специализацией для типов std::vector (обратите внимание, она наследуется от true_type). Поэтому когда мы передаем тип std::vector в структуру is_stdvector, компилятор выбирает специализированную версию шаблона. Если мы передаем тип данных, отличный от std::vector, используется универсальная версия, которая наследуется от false_type.

Перед boost::false_type и boost::true_type в нашем свойстве типа нет ключевого слова public, потому что мы используем ключевое слово struct, а оно применяет публичное наследование по умолчанию.

Дополнительно...

Если вы используете совместимые с C++11 компиляторы, то для создания собственных свойств типов вам доступны типы std::true_type и std::false_type, объявленные в заголовке <type_traits>.

Начиная с C++17 в стандартной библиотеке есть псевдоним типа std::bool_constant<true_or_false>, который вы можете использовать для удобства.

Как обычно, Boost-версии классов и функций являются более переносимыми, поскольку их можно использовать в компиляторах для C++11.

См. также

- Почти во всех рецептах из этой главы используются свойства типов. Обратитесь к документации Boost.TypeTraits, чтобы найти дополнительные примеры и информацию на странице http://boost.org/libs/type_traits;
- *см.* предыдущий рецепт, чтобы получить больше информации об интегральных константах и о том, как можно реализовать true_type и false_type с нуля.

Выбор оптимального оператора для параметра шаблона

Представьте, что мы работаем с классами от разных поставщиков, которые реализуют разный набор арифметических операций и имеют конструкторы от целого числа. Мы хотим создать функцию, которая увеличивает на 1 любой класс, который передается ей.

Также мы хотим, чтобы эта функция была эффективной! Посмотрите на приведенный ниже код:

```
template <class T>
void inc(T& value) {
    // TODO:
    // вызвать ++value
    // или вызвать value++
    // или value += T(1);
```

```
    // или value = value + T(1);
}
```

Подготовка

Требуются базовые знания шаблонов C++.

Как это делается...

Выбор оптимального оператора можно сделать во время компиляции. Этого можно добиться с помощью библиотеки `Boost.TypeTraits`, как показано ниже.

1. Давайте начнем с создания правильных функциональных объектов:

```
namespace detail {
    struct pre_inc_functor {
    template <class T>
        void operator()(T& value) const {
           ++ value;
        }
    };

    struct post_inc_functor {
    template <class T>
        void operator()(T& value) const {
            value++;
        }
   };

   struct plus_assignable_functor {
   template <class T>
       void operator()(T& value) const {
           value += T(1);
       }
   };

   struct plus_functor {
   template <class T>
       void operator()(T& value) const {
           value = value + T(1);
       }
   };
}
```

2. После этого нам понадобится несколько свойств типов:

```
#include <boost/type_traits/conditional.hpp>
#include <boost/type_traits/has_plus_assign.hpp>
#include <boost/type_traits/has_plus.hpp>
#include <boost/type_traits/has_post_increment.hpp>
#include <boost/type_traits/has_pre_increment.hpp>
```

3. Мы готовы выбрать правильный функтор и использовать его:

```
template <class T>
void inc(T& value) {
    // вызвать ++value
    // или вызвать value++
    // или value += T(1);
    // или value = value + T(1);

    typedef detail::plus_functor step_0_t;

    typedef typename boost::conditional<
      boost::has_plus_assign<T>::value,
      detail::plus_assignable_functor,
      step_0_t
    >::type step_1_t;

    typedef typename boost::conditional<
      boost::has_post_increment<T>::value,
      detail::post_inc_functor,
      step_1_t
    >::type step_2_t;

    typedef typename boost::conditional<
      boost::has_pre_increment<T>::value,
      detail::pre_inc_functor,
    step_2_t
    >::type step_3_t;

    step_3_t() // Конструктор функтора по умолчанию.
      (value); // Вызываем operator() у функтора.
}
```

Как это работает...

Вся магия осуществляется через метафункцию `conditional<bool Condition, class T1, class T2>`. Когда значение `true` передается в метафункцию в качестве первого параметра, она возвращает `T1` через `::type typedef`. Когда значение `false` передается в метафункцию в качестве первого параметра, она возвращает `T2` через `::type typedef`. Она действует как своего рода условный тернарный оператор времени компиляции.

Так, тип `step0_t` содержит метафункцию `detail::plus_functor`, а тип `step1_t` включает `step0_t` или `detail::plus_assignable_functor`. Тип `step2_t` содержит `step1_t` или `detail::post_inc_functor`, тип `step3_t` – `step2_t` или `detail::pre_inc_functor`.

Содержимое каждого `step*_t typedef` выводится с использованием свойства типов.

Дополнительно...

`conditional` существует в C++11, ее можно найти в заголовке `<type_traits>` в пространстве имен `std::`. В Boost есть несколько версий этой функции в разных библиотеках; например, в `Boost.MPL` есть функция `boost::mpl::if_c`, которая

действует точно так же, как boost::conditional, а еще версия boost::mpl::if_ (без буквы c на конце), которая вызывает ::type для своего первого аргумента шаблона. Если первый аргумент наследуется от boost::true_type, метафункция возвращает второй аргумент во время вызова ::type. В противном случае она возвращает последний шаблонный параметр. Мы можем переписать нашу функцию inc(), чтобы использовать Boost.MPL:

```
#include <boost/mpl/if.hpp>

template <class T>
void inc_mpl(T& value) {
    typedef detail::plus_functor step_0_t;

    typedef typename boost::mpl::if_<
      boost::has_plus_assign<T>,
      detail::plus_assignable_functor,
      step_0_t
    >::type step_1_t;

    typedef typename boost::mpl::if_<
      boost::has_post_increment<T>,
      detail::post_inc_functor,
      step_1_t
    >::type step_2_t;

    typedef typename boost::mpl::if_<
      boost::has_pre_increment<T>,
      detail::pre_inc_functor,
      step_2_t
    >::type step_3_t;

    step_3_t() // Конструктор функтора по умолчанию.
      (value); // Вызываем operator() функтора.
}
```

В C++17 есть конструкция if constexpr, которая значительно упрощает предыдущий пример:

```
template <class T>
void inc_cpp17(T& value) {
    if constexpr (boost::has_pre_increment<T>()) {
        ++value;
    } else if constexpr (boost::has_post_increment<T>()) {
        value++;
    } else if constexpr(boost::has_plus_assign<T>()) {
        value += T(1);
    } else {
        value = value + T(1);
    }
}
```

В предыдущем примере мы использовали альтернативный способ получения значения из свойств типа. Интегральные константы в стандартной библиотеке C++ Boost.MPL и Boost.TypeTraits имеют оператор преобразования constexpr. Это означает, что экземпляр std::true_type можно преобразовать в значение true. При добавлении круглых скобок () или фигурных скобок C++11 {} создается экземпляр класса boost::has_pre_increment<T>, который можно преобразовать в значения true или false.

См. также

- Рецепт «*Активация использования шаблонных функций для интегральных типов*»;
- рецепт «*Отключение использования шаблонных функций для действительных типов*»;
- документация по библиотеке Boost.TypeTraits содержит полный список доступных метафункций. Перейдите по ссылке http://boost.org/libs/type_traits;
- в рецептах главы 8 «*Метапрограммирование*» приводятся дополнительные примеры использования библиотеки Boost.MPL. Если вы чувствуете себя уверенно с шаблонами, можете также попробовать прочитать документацию по ссылке http://boost.org/libs/mpl.

Получение типа выражения в C++03

В предыдущих рецептах мы видели примеры использования boost::bind. Он может быть полезным инструментом в мире стандартов, предшествующих C++11, но трудно сохранить результат функции boost::bind в переменную в C++03.

```
#include <functional>
#include <boost/bind.hpp>

const ??? var = boost::bind(std::plus<int>(), _1, _1);
```

В C++11 можно использовать ключевое слово auto вместо ???, и это сработает. А что делать в C++03?

Подготовка

Знание ключевых слов auto и decltype из C++11 может помочь вам понять этот рецепт.

Как это делается...

Нам понадобится библиотека Boost.Typeof для получения возвращаемого выражения типа:

```
#include <boost/typeof/typeof.hpp>

BOOST_AUTO(var, boost::bind(std::plus<int>(), _1, _1));
```

Как это работает...

Макрос BOOST_AUTO создает переменную с именем var, а выражение для инициализации переменной передается в качестве второго аргумента. Тип var определяется по типу выражения.

Дополнительно...

Опытный читатель заметит, что в C++11 существуют и другие ключевые слова для определения типа выражения. В Boost.Typeof также есть макросы и для них. Давайте посмотрим на приведенный ниже C++11:

```
typedef decltype(0.5 + 0.5f) type;
```

Используя библиотеку Boost.Typeof, предыдущий код можно написать так:

```
typedef BOOST_TYPEOF(0.5 + 0.5f) type;
```

decltype(expr) выводит и возвращает тип выражения:

```
template<class T1, class T2>
auto add(const T1& t1, const T2& t2) ->decltype(t1 + t2) {
    return t1 + t2;
};
```

Используя Boost.Typeof, предыдущий код можно записать так:

```
// Длинный и переносимый способ:
template<class T1, class T2>
struct result_of {
    typedef BOOST_TYPEOF_TPL(T1() + T2()) type;
};

template<class T1, class T2>
typename result_of<T1, T2>::type add(const T1& t1, const T2& t2) {
    return t1 + t2;
};

// ... или ...

// Укороченная версия, которая может вывести из строя некоторые компиляторы.
template<class T1, class T2>
BOOST_TYPEOF_TPL(T1() + T2()) add(const T1& t1, const T2& t2) {
    return t1 + t2;
};
```

В C++11 имеется специальный синтаксис для указания типа возвращаемого значения в конце объявления функции. К сожалению, его нельзя сэмулировать в C++03, поэтому мы не можем использовать переменные t1 и t2 в макросе.

Вы можете свободно использовать результаты функций BOOST_TYPEOF() в шаблонах и в любых других выражениях времени компиляции:

```
#include <boost/static_assert.hpp>
#include <boost/type_traits/is_same.hpp>
BOOST_STATIC_ASSERT((
    boost::is_same<BOOST_TYPEOF(add(1, 1)), int>::value
));
```

Однако, к сожалению, эта магия не всегда работает без посторонней помощи. Например, пользовательские шаблонные классы не всегда обнаруживаются, поэтому приведенный ниже код на некоторых компиляторах может не работать:

```
namespace readers_project {
    template <class T1, class T2, class T3>
    struct readers_template_class{};
}

#include <boost/tuple/tuple.hpp>

typedef
    readers_project::readers_template_class<int, int, float>
readers_template_class_1;

typedef BOOST_TYPEOF(boost::get<0>(
    boost::make_tuple(readers_template_class_1(), 1)
)) readers_template_class_deduced;

BOOST_STATIC_ASSERT((
    boost::is_same<
        readers_template_class_1,
        readers_template_class_deduced
    >::value
));
```

В таких ситуациях вы можете протянуть `Boost.Type` руку помощи и зарегистрировать ваш шаблонный класс:

```
BOOST_TYPEOF_REGISTER_TEMPLATE(
        readers_project::readers_template_class /* имя класса */,
        3 /* количество шаблонных параметров */
)
```

Однако три самых популярных компилятора правильно определили тип из примера даже без `BOOST_TYPEOF_REGISTER_TEMPLATE` и C++11.

См. также

В официальной документации `Boost.Typeof` содержится еще больше примеров. Перейдите по ссылке http://boost.org/libs/typeof.

Глава 5

Многопоточность

Темы, которые мы рассмотрим в этой главе:

- создание потока выполнения;
- синхронизация доступа к общему ресурсу;
- быстрый доступ к общему ресурсу с использованием атомарных операций;
- создание класса `work_queue`;
- блокировка «несколько читателей – один писатель»;
- создание переменных, уникальных для каждого потока;
- прерывание потока;
- манипулирование группой потоков;
- безопасная инициализация общей переменной;
- захват нескольких мьютексов.

Вступление

В этой главе мы займемся потоками и всем, что с ними связано. Базовые знания о многопоточности приветствуются.

Многопоточность означает, что в одном процессе существует несколько потоков выполнения.

Потоки могут совместно использовать ресурсы процесса и иметь собственные ресурсы. Эти потоки выполнения могут происходить независимо на разных процессорах, что позволяет нам писать более быстрые и отзывчивые программы. Библиотека `Boost.Thread` обеспечивает унификацию интерфейсов операционных систем для работы с потоками. Это не header-only библиотека, поэтому все примеры из этой главы должны линковаться с библиотеками `libboost_thread` и `libboost_system`.

Создание потока выполнения

В современных многоядерных компьютерах для достижения максимальной производительности (или просто для обеспечения отзывчивого пользовательского интерфейса) программы обычно используют несколько потоков выполнения.

Вот мотивирующий пример, в котором мы создаем и заполняем большой файл в потоке, рисующем пользовательский интерфейс:

```
#include <cstddef> // для std::size_t

bool is_first_run();

// Функция, которая выполняется в течение длительного времени.
void fill_file(char fill_char, std::size_t size, const char* filename);

// Вызывается в потоке, который отрисовывает пользовательский интерфейс:
void example_without_threads() {
    if (is_first_run()) {
        // Этот код будет выполняться длительное время, в течение которого
        // пользовательский интерфейс зависнет…
        fill_file(0, 8 * 1024 * 1024, "save_file.txt");
    }
}
```

Нужно исправить пример и сделать так, чтобы пользовательский интерфейс не «подвисал».

Подготовка

Этот рецепт требует знания `boost::bind` или `std::bind`.

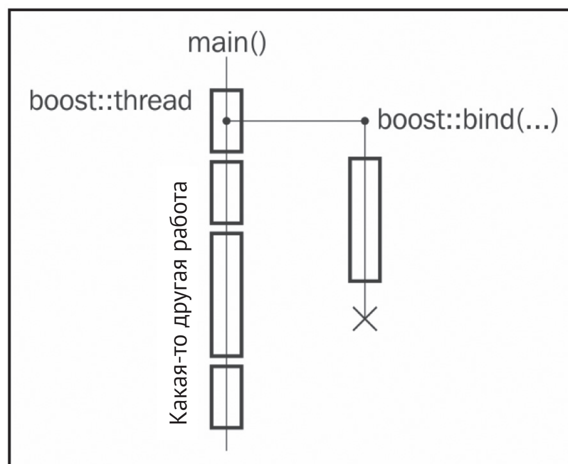
Как это делается...

Запуск потока выполнения никогда не был проще:

```
#include <boost/thread.hpp>

// Вызывается в потоке, который отрисовывает пользовательский интерфейс:
void example_with_threads() {
    if (is_first_run()) {
        boost::thread(boost::bind(
            &fill_file,
            0,
            8 * 1024 * 1024,
            "save_file.txt"
        )).detach();
    }
}
```

Как это работает...

Переменная `boost::thread` принимает функциональный объект, который можно вызывать без параметров (мы предоставили один из них с помощью `boost::bind`), и создает отдельный поток выполнения. Функциональный объект копируется в созданный поток выполнения и запускается в нем. Возвращаемое значение функционального объекта игнорируется.



Мы используем библиотеку Boost.Thread четвертой версии во всех рецептах (макрос BOOST_THREAD_VERSION выставлен в значение 4 во всех примерах). Важные различия между версиями библиотеки Boost.Thread будут освещены.

После этого мы вызываем функцию detach(), которая выполняет следующие действия:

- поток выполнения отделяется от переменной boost::thread, но продолжает выполнение;
- переменная boost::thread переходит в состояние Not-A-Thread.

Без вызова функции detach() деструктор boost::thread заметит, что он по-прежнему содержит поток ОС, и вызовет std::terminate. Он завершит нашу программу без вызова деструкторов, освобождения ресурсов и другой очистки.

Конструктор boost::thread без параметров создает переменную в состоянии Not-A-Thread, отдельный поток выполнения не создается.

Дополнительно...

Если мы хотим дождаться завершения выполнения записи файла, нам нужно вызвать функцию-член join:

```
void example_with_joining_threads() {
    if (is_first_run()) {
        boost::thread t(boost::bind(
            &fill_file,
            0,
            8 * 1024 * 1024,
            "save_file.txt"
        ));

        // Выполняем какую-то работу.
```

```
        // ...

        // Ждем завершения потока.
        t.join();
    }
}
```

После этого переменная boost::thread переходит в состояние Not-A-Thread, а ее деструктор не вызывает std::terminate.

Помните, что нужно выполнять функцию-член join или detach для потока до вызова деструктора. В противном случае ваша программа завершит работу!

При выставлении BOOST_THREAD_VERSION=2 деструктор boost::thread начинает автоматически вызывать функцию detach(), что не приводит к std::terminate. Но это нарушает совместимость с std::thread и преподнесет вам много неприятных сюрпризов, когда ваш проект будет переезжать на использование стандартной библиотеки потоков C++ или когда BOOST_THREAD_VERSION=2 перестанет поддерживаться. Версия 4 Boost.Thread является более явной и строгой, что считается предпочтительным в языке C++.

Помните, что std::terminate() вызывается, и когда любое исключение, не относящееся к типу boost::thread_interrupted, выходит за границы функционального объекта, переданного конструктору boost::thread.

Существует очень полезная RAII-обертка для boost::thread, позволяющая эмулировать поведение BOOST_THREAD_VERSION=2. Она называется boost::scoped_thread<T>, где T может быть одним из следующих классов:

- boost::interrupt_and_join_if_joinable – прерывает и вызывает join потоку при уничтожении;
- boost::join_if_joinable – вызывает потоку при уничтожении;
- boost::detach: – вызывает detach потоку при уничтожении.

Вот краткий пример использования:

```
#include <boost/thread/scoped_thread.hpp>

void some_func();

void example_with_raii() {
    boost::scoped_thread<boost::join_if_joinable> t(
        boost::thread(&some_func)
    );

    // Функция-член join будет вызвана при выходе из области видимости.
}
```

Класс boost::thread был принят как часть стандарта C++11, и его можно найти в заголовке <thread> в пространстве имен std::. Между Boost версии 4 и стандартной библиотекой C++11 нет особой разницы. Тем не менее boost::thread доступен на компиляторах C++03, поэтому его использование более переносимо.

Существует очень веская причина для вызова `std::terminate` вместо `join` по умолчанию в деструкторе! Языки C и C++ часто используются в жизненно важном программном обеспечении. Такое ПО контролируется другим программным обеспечением под названием сторожевой таймер (watchdog). Эти сторожевые таймеры предназначены для быстрого перезапуска приложения, если с ним что-то не так. Они с легкостью могут обнаружить, что приложение экстренно завершило работу, но не всегда могут найти взаимоблокировки или обнаруживают их со значительными задержками. Так, для программного обеспечения дефибриллятора безопаснее экстренно завершить работу, чем висеть на `join()` несколько секунд в ожидании реакции сторожевого таймера. Имейте это в виду при разработке таких приложений.

См. также

- Все рецепты в этой главе используют библиотеку `Boost.Thread`. Вы можете читать дальше, чтобы получить больше информации о ней;
- официальная документация содержит полный список методов `boost::thread` и замечания относительно их доступности в стандартной библиотеке C++11. Перейдите по ссылке http://boost.org/libs/thread;
- рецепт «*Прерывание потока*» даст вам представление о том, что делает класс `boost:interrupt_and_join_if_joinable`.

Синхронизация доступа к общему ресурсу

Теперь, когда мы знаем, как запускать потоки выполнения, нам нужен доступ к общим ресурсам из разных потоков:

```
#include <cassert>
#include <cstddef>
#include <iostream>

// В предыдущем рецепте мы подключили заголовочный файл <boost/thread.hpp>,
// который включает в себя все классы и методы библиотеки Boost.Thread.
// Приведенный ниже заголовок включает в себя только класс boost::thread
#include <boost/thread/thread.hpp>

int shared_i = 0;

void do_inc() {
    for (std::size_t i = 0; i < 30000; ++i) {
        const int i_snapshot = ++shared_i;
        // Делаем что-то с i_snapshot.
        // ...
    }
}

void do_dec() {
    for (std::size_t i = 0; i < 30000; ++i) {
        const int i_snapshot = --shared_i;
```

```
        // Делаем что-то с i_snapshot.
        // ...
    }
}
void run() {
    boost::thread t1(&do_inc);
    boost::thread t2(&do_dec);

    t1.join();
    t2.join();

    assert(global_i == 0); // Ой! Ай! Бооооль!!!
    std::cout << "shared_i == " << shared_i;
}
```

Это "Ой! Ай! Бооооль!" здесь неспроста. Для некоторых это может быть сюрпризом, но есть большая вероятность того, что переменная shared_i не будет равна 0:

```
shared_i == 19567
```

Современные компиляторы и процессоры имеют огромное количество разных хитрых оптимизаций, которые могут нарушить предыдущий код. Мы не будем обсуждать их здесь, но в разделе «См. также» есть полезная ссылка, которая кратко описывает их.

Ситуация становится еще хуже в случаях, когда общий ресурс является нетривиальным классом; ошибки сегментации, падения приложения и утечки памяти могут (и будут) происходить.

Нам нужно изменить код так, чтобы только один поток модифицировал переменную shared_i в один момент времени.

Подготовка

Для этого рецепта рекомендуется иметь базовые знания многопоточности.

Как это делается...

Давайте посмотрим, как можно исправить предыдущий пример и сделать переменную shared_i равной 0 в конце выполнения.

1. Прежде всего нужно создать **мьютекс**:

```
#include <boost/thread/mutex.hpp>
#include <boost/thread/locks.hpp>

int shared_i = 0;
boost::mutex i_mutex;
```

2. Поместите все операции, которые изменяют или получают данные из переменной shared_i, между

```
{ // Начало критической секции
      boost::lock_guard<boost::mutex> lock(i_mutex);
```

и:

```
} // Конец критической секции
```

Вот как это должно выглядеть:

```
void do_inc() {
    for (std::size_t i = 0; i < 30000; ++i) {
        int i_snapshot;
        { // Начало критической секции
            boost::lock_guard<boost::mutex> lock(i_mutex);
            i_snapshot = ++shared_i;
        } // Конец критической секции

        // Делаем что-то с i_snapshot.
        // ...
    }
}

void do_dec() {
    for (std::size_t i = 0; i < 30000; ++i) {
        int i_snapshot;
        { // Начало критической секции
            boost::lock_guard<boost::mutex> lock(i_mutex);
            i_snapshot = --shared_i;
        } // Конец критической секции
        // Делаем что-то с i_snapshot.
        // ...
    }
}
```

Как это работает...

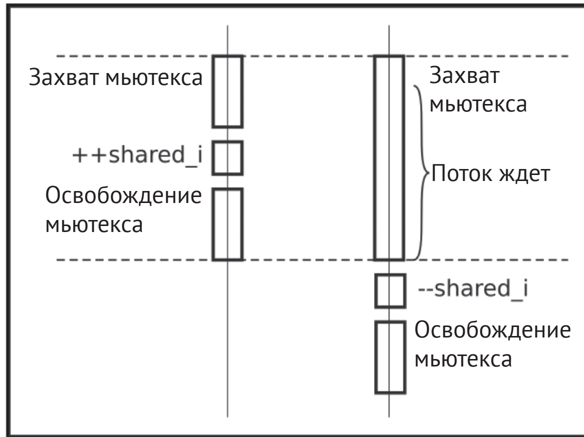
Класс `boost::mutex` заботится обо всем, что касается синхронизации. Когда поток пытается захватить его с помощью переменной `boost::lock_guard<boost:mutex>` и нет другого потока, захватившего этот же `boost::mutex`, поток успешно получает уникальный доступ к блоку кода. Если какой-либо поток уже захватил `boost::mutex`, другой поток, пытающийся захватить `boost::mutex`, будет приостановлен до тех пор, пока поток, захвативший `boost::mutex`, не освободит его. Все операции захвата/освобождения используют специальные машинные инструкции, чтобы:

- изменения данных в критической секции были видны всем потокам;
- процессор не переупорядочивал инструкции;
- компилятор не переупорядочивал инструкции;
- компилятор не удалял записи в переменные, которые не читаются в текущем потоке;
- не происходила целая куча других неприятных вещей, связанных с компилятором и особенностями архитектуры.

Если у вас есть переменная, которую читают из разных потоков, и по крайней мере один поток изменяет эту переменную, то обычно весь код, который ее использует, должен рассматриваться как критическая секция и защищаться мьютексом.

`boost::lock_guard` – очень простой RAII-класс, который хранит ссылку на мьютекс, захватывает его в конструкторе и освобождает в деструкторе.

В предыдущем примере мы использовали фигурные скобки для управления временем жизни переменной `lock`. При достижении закрывающей скобки вызывается деструктор для переменной `lock` и мьютекс будет освобожден. Даже если в критической секции происходит какое-то исключение, мьютекс будет правильно освобожден, т. к. отработает деструктор переменной `lock`.



Если вы инициализируете общую переменную, а затем создаете потоки, которые только читают ее, то никакого мьютекса или другой синхронизации не требуется.

Дополнительно...

Захват мьютекса – потенциально очень медленная операция, которая может остановить выполнение вашего кода на долгое время, пока какой-либо другой поток не освободит мьютекс. Постарайтесь делать критические разделы как можно меньшего размера, так чтобы их было как можно меньше в вашем коде.

Давайте посмотрим, как некоторые операционные системы обрабатывают захват мьютекса на многоядерном процессоре. Когда `thread #1`, работающий на ядре 1, пытается захватить мьютекс, который уже захвачен другим потоком, ОС останавливает `thread #1` до освобождения мьютекса. Остановленный поток не использует ресурсы процессора, поэтому ОС может начать выполнять другой поток на ядре 1. Теперь у нас есть поток, работающий на ядре 1. Какой-то поток освобождает мьютекс, и теперь ОС должна возобновить выполнение `thread #1`. Она возобновляет его выполнение на свободном в данный момент ядре, например на ядре 2.

Это приводит к промахам кеша ЦП, поэтому код после освобождения мьютекса может работать немного медленнее. Обычно все не так плохо, потому что хорошая ОС старается возобновить поток на том же процессоре, который использовался для этого потока ранее. К сожалению, такие оптимизации не всегда возможны. Уменьшите количество критических секций и их размеры, чтобы сократить вероятность приостановки потока и промахов кеша.

Не пытайтесь захватить переменную `boost::mutex` дважды в одном и том же потоке; это приведет к непредсказуемым результатам (например, к **взаимной блокировке**). Если требуется захватить мьютекс несколько раз из одного потока, используйте класс `boost::recursive_mutex` из заголовочного файла `<boost/thread/recursive_mutex.hpp>`. Многократный захват этого мьютекса из одного и того же потока не приводит к взаимной блокировке. `boost::recursive_mutex` заканчивает критическую секцию только после того, как мьютекс будет освобожден столько же раз, сколько был захвачен. Избегайте использования класса `boost::recursive_mutex`, когда его функциональность не требуется, потому что этот класс медленнее `boost::mutex` и его использование обычно указывает на плохую архитектуру приложения.

Классы `boost::mutex`, `boost::recursive_mutex` и `boost::lock_guard` были приняты в стандартную библиотеку C++11, и вы можете найти их в заголовочном файле `<mutex>` в пространстве имен `std::`. Версия Boost может иметь расширения (которые отмечены в официальной документации как *EXTENSION*) и обеспечивать лучшую переносимость, поскольку методы и классы из Boost можно использовать даже на компиляторах, не поддерживающих C++11.

См. также

- Приведенный далее рецепт даст вам идеи, как сделать текущий рецепт намного быстрее (и короче);
- прочтите первый рецепт из этой главы, чтобы получить больше информации о классе `boost::thread`. Официальная документация по адресу http://boost.org/libs/thread также может быть полезной;
- чтобы получить больше информации о том, почему первый пример не работает и как многоядерные процессоры работают с общими ресурсами, см. статью на странице http://www.rdrop.com/users/paulmck/scalability/paper/whymb.2010.07.23a.pdf. Обратите внимание, что это крайне сложная тема.

Быстрый доступ к общему ресурсу с использованием атомарных операций

В предыдущем рецепте мы увидели, как безопасно получить доступ к общему ресурсу из разных потоков. Но в том рецепте мы делали до двух системных вызовов (при захвате и освобождении мьютекса), чтобы просто получить значение из целого числа:

```
{ // Начало критической секции
  boost::lock_guard<boost::mutex> lock(i_mutex);
  i_snapshot = ++ shared_i;
} // Конец критической секции
```

Выглядит коряво и медленно! Можно ли улучшить код из предыдущего рецепта?

Подготовка

Чтение прошлого рецепта – все, что вам нужно, чтобы начать. Базовые знания о многопоточности приветствуются.

Как это делается...

Давайте посмотрим, как улучшить наш предыдущий пример:

1. Теперь нам нужны разные заголовки:

```
#include <cassert>
#include <cstddef>
#include <iostream>

#include <boost/thread/thread.hpp>
#include <boost/atomic.hpp>
```

2. Требуется изменить тип `shared_i`:

```
boost::atomic<int> shared_i(0);
```

3. Удаляем все переменные `boost::lock_guard`:

```
void do_inc() {
   for (std::size_t i = 0; i < 30000; ++i) {
       const int i_snapshot = ++shared_i;
       // Делаем что-то с i_snapshot.
       // ...
   }
}

void do_dec() {
   for (std::size_t i = 0; i < 30000; ++i) {
       const int i_snapshot = --shared_i;

       // Делаем что-то i_snapshot.
       // ...
  }
}
```

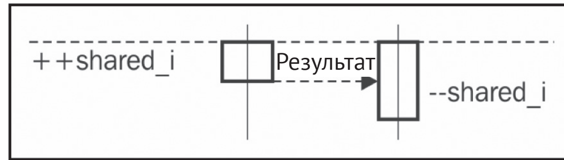
4. Готово! Теперь все работает:

```
int main() {
    boost::thread t1(&do_inc);
    boost::thread t2(&do_dec);

    t1.join();
    t2.join();

    assert(shared_i == 0);
    std::cout << "shared_i == " << shared_i << std::endl;

    assert(shared_i.is_lock_free());
}
```

Как это работает...

Процессоры предоставляют **атомарные операции**, работа которых не может быть нарушена другими процессорами или ядрами процессора. Для системы эти операции происходят мгновенно. Библиотека `Boost.Atomic` предоставляет классы, которые взаимодействуют с компилятором, чтобы отключить оптимизации, которые могут нарушить многопоточную работу с переменной,

и обеспечивают унифицированный переносимый интерфейс для работы с платформоспецифичными атомарными операциями. Если две атомарные операции работают с одной и той же ячейкой памяти и запускаются одновременно из разных потоков, одна из операций будет ждать, пока не закончится другая, а затем использует ее результат для вычислений.



Другими словами, использовать переменные `boost::atomic<>` из разных потоков одновременно – это безопасно. Каждая операция с атомарной переменной рассматривается системой как отдельная транзакция. Ряд операций над атомарными переменными обрабатывается системой как ряд независимых транзакций:

```
--shared_i; // Транзакция #1
//Другие потоки могут изменить значение `shared_i`!!
++shared_i; // Транзакция #2
```

Никогда не пренебрегайте синхронизацией для переменной, которая модифицируется из нескольких потоков. Даже если переменная имеет тип `bool`, и все, что вы делаете, – это читаете или пишете значения `true`/`false` в нее! Компилятор и процессор имеют право оптимизировать операции хранения и чтения, ломая ваш код миллионами заковыристых способов. Угадайте, кого накажет хороший работодатель за такую неисправность? (Компилятор – неверный ответ на этот вопрос!)

Дополнительно...

Библиотека `Boost.Atomic` может работать только с типами POD; в противном случае поведение будет неопределенным. Некоторые платформы и процессоры не обеспечивают атомарные операции для некоторых типов, поэтому `Boost.Atomic` эмулирует атомарное поведение, используя класс `boost::mutex`. Атомарный тип не использует `boost::mutex`, если специфический для типа макрос установлен в значение 2:

```
#include <boost/static_assert.hpp>
BOOST_STATIC_ASSERT(BOOST_ATOMIC_INT_LOCK_FREE == 2);
```

Функция-член `boost::atomic<T>::is_lock_free` не годится для проверок времени компиляции, но может обеспечить более читаемый синтаксис, когда проверки времени выполнения достаточно:

```
assert(shared_i.is_lock_free());
```

Атомарные операции работают намного быстрее, чем мьютексы, но все же намного медленнее, чем неатомарные. Если мы сравним время выполнения рецепта с мьютексами и примера в этом рецепте, то увидим разницу минимум раза в 4.

У всех известных автору книги реализаций стандартной библиотеки C++ и у Boost были проблемы с атомарными операциями. У всех! Не пишите собственные функции и классы атомарных операций. Если вы думаете, что ваша собственная реализация атомарной операции будет лучше и вы хотите потратить время – напишите ее, проверьте с помощью специальных инструментов и подумайте еще раз. Делайте это до тех пор, пока не поймете, что вы ошибаетесь.

Совместимые с C++11 компиляторы должны иметь все атомарные классы, псевдонимы типов и макросы в заголовке `<atomic>` в пространстве имен `std::`. Специфичные для компилятора реализации `std::atomic` могут работать быстрее, чем Boost-версия.

См. также

Официальная документация на странице http://boost.org/libs/atomic может предоставить дополнительные примеры и теоретическую информацию по этой теме.

Создание класса work_queue

Давайте вызовем функциональный объект, который не принимает аргументов (будем называть его «задачей» для краткости).

```
typedef boost::function<void()> task_t;
```

Теперь представьте ситуацию, когда у нас есть потоки, которые публикуют задачи, и потоки, которые выполняют опубликованные задачи. Нам нужно спроектировать класс, один экземпляр которого может одновременно и безопасно использоваться обоими типами этих потоков. Этот класс должен иметь функции для:

- получения задачи или ожидания публикации задачи;
- неблокирующего получения задачи (или возврата пустой задачи, если задач не осталось);
- публикации задач.

Подготовка

Убедитесь, что вы чувствуете себя комфортно с `boost::thread` или `std::thread`, знаете основы мьютексов и что такое `boost::function` или `std::function`.

Как это делается...

Класс, который мы собираемся реализовать, близок по функциональности к `std::queue<task_t>`, но при этом с ним безопасно работать из разных потоков. Давайте начнем.

1. Нам нужны следующие заголовочные файлы и члены класса:

```
#include <deque>
#include <boost/function.hpp>
#include <boost/thread/mutex.hpp>
```

```
#include <boost/thread/locks.hpp>
#include <boost/thread/condition_variable.hpp>

class work_queue {
public:
    typedef boost::function<void()> task_type;

private:
    std::deque<task_type> tasks_;
    boost::mutex tasks_mutex_;
    boost::condition_variable cond_;
```

2. Функция для публикации (помещения задачи в очередь) должна выглядеть так:

```
public:
    void push_task(const task_type& task) {
        boost::unique_lock<boost::mutex> lock(tasks_mutex_);
        tasks_.push_back(task);
        lock.unlock();

        cond_.notify_one();
}
```

3. Неблокирующая функция для получения опубликованной или пустой задачи, если задач не осталось:

```
task_type try_pop_task() {
    task_type ret;
    boost::lock_guard<boost::mutex> lock(tasks_mutex_);
    if (!tasks_.empty()) {
        ret = tasks_.front();
        tasks_.pop_front();
    }

    return ret;
}
```

4. Функция для получения размещенной задачи или для ожидания, когда задача будет опубликова над другим потоком:

```
  task_type pop_task() {
      boost::unique_lock<boost::mutex> lock(tasks_mutex_);
      while (tasks_.empty()) {
          cond_.wait(lock);
      }

      task_type ret = tasks_.front();
      tasks_.pop_front();

      return ret;
  }
};
```

Вот как можно использовать класс `work_queue`:

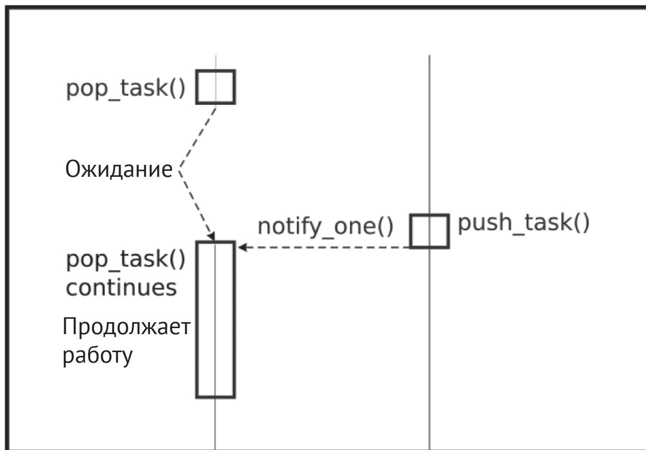
```
#include <boost/thread/thread.hpp>

work_queue g_queue;

void some_task();
const std::size_t tests_tasks_count = 3000 /*000*/;

void pusher() {
    for (std::size_t i = 0; i < tests_tasks_count; ++i) {
        g_queue.push_task(&some_task);
    }
}

void popper_sync() {
    for (std::size_t i = 0; i < tests_tasks_count; ++i) {
        work_queue::task_type t = g_queue.pop_task();
        t(); // Выполнение задачи.
    }
}

int main() {
    boost::thread pop_sync1(&popper_sync);
    boost::thread pop_sync2(&popper_sync);
    boost::thread pop_sync3(&popper_sync);

    boost::thread push1(&pusher);
    boost::thread push2(&pusher);
    boost::thread push3(&pusher);

    // Ожидаем публикации всех задач.
    push1.join();
    push2.join();
    push3.join();
    g_queue.flush();

    // Ожидаем извлечения всех задач
    pop_sync1.join();
    pop_sync2.join();
    pop_sync3.join();

    // Проверяем, что никаких заданий не осталось, и продолжаем выполнение без блокировки.
    assert(!g_queue.try_pop_task());

    g_queue.push_task(&some_task);

    // Проверяем, что задача есть, и продолжаем выполнение  без блокировки.
    assert(g_queue.try_pop_task());
}
```

Как это работает...

В этом примере мы видим новый RAII-класс `boost::unique_lock`. Фактически это класс `boost::lock_guard` с дополнительной функциональностью для явного захвата и освобождения мьютексов.

Вернемся к нашему классу `work_queue`. Давайте начнем с функции `pop_task()`. Сначала мы захватываем мьютекс и проверяем наличие доступных задач. Если задача есть, мы ее возвращаем; в противном случае вызывается `cond_.wait(lock)`. Этот метод атомарно освобождает мьютекс и приостанавливает выполнение потока, пока какой-либо другой поток не уведомит текущий поток.

Теперь давайте посмотрим на метод `push_task`. В нем мы также захватываем мьютекс, помещаем задачу в `tasks_.queue`, освобождаем мьютекс и вызываем метод `cond_.notify_one()`, который пробуждает поток (если таковой имеется), ожидающий на `cond_.wait(lock)`. Итак, после этого, если какой-то поток ожидает задачу в функции `pop_task()`, поток продолжит свое выполнение, вызовет функцию `lock.lock()` глубоко внутри `cond_.wait(lock)` и проверит `tasks_empty()` в цикле `while`. Поскольку мы только что добавили задачу в `tasks_`, мы выйдем из цикла `while`, освободим `mutex` в деструкторе переменной `lock` и вернем задачу.



Вы должны проверять условия в цикле, а не просто в операторе `if`! Оператор `if` приводит к появлению ошибок, так как иногда операционная система может пробуждать потоки без каких-либо уведомлений от пользователя.

Дополнительно...

Обратите внимание, что мы явно освободили мьютекс перед вызовом функции `notify_one()`. Однако и без освобождения наш пример бы работал.

Но в этом случае проснувшийся поток может быть снова усыплен ОС во время попытки вызвать функцию `lock.lock()` глубоко внутри `cond_wait(lock)`, что приводит к большему количеству переключений контекста и ухудшению производительности.

Так без явного освобождения мьютекса этот пример выполняется в течение 7 секунд, при выставлении переменной `tests_tasks_count` в значение `3000000`:

```
$time -f E ./work_queue
0:07.38
```

При явном освобождении мьютекса этот пример выполняется в течение 5 секунд:

```
$ time -f E ./work_queue
0:05.39
```

Вы также можете уведомить все потоки, ожидающие на условной переменной, с помощью функции `cond_.notify_all()`.

У некоторых чрезвычайно экзотических операционных систем была довольно редкая проблема с вызовом `notify_one()` вне критической секции в Boost до появления версии 1.64: https://github.com/boostorg/thread/pull/105. Вряд ли вы когда-нибудь будете работать с такими платформами. Но, во всяком случае чтобы избежать проблем на этих платформах, вы можете добавить функцию `flush()` в класс `work_queue`, который захватывает мьютекс и вызывает функцию `notify_all()`:

```
void flush () {
    boost::lock_guard<boost::mutex> lock(tasks_mutex_);
    cond_.notify_all();
}
```

Вызовите функцию `flush()`, когда закончите работать с очередью, чтобы принудительно разбудить все потоки.

В стандарте C++11 есть `std::condition_variable`, объявленный в заголовочном файле `<condition_variable>`, и `std::unique_lock`, объявленный в `<mutex>`. Используйте версию Boost, если применяете компилятор C++03 или собираетесь воспользоваться расширениями Boost.

Класс `work_queue` можно значительно улучшить, добавив поддержку rvalue-ссылок и вызвав `std::move(tasks_.front())`. Это сделает код в критической секции намного быстрее, что приведет к меньшему количеству приостановок и пробуждений потоков, меньшему количеству промахов кеша и намного лучшей производительности.

См. также

- Первые три рецепта в этой главе содержат много полезной информации о библиотеке `Boost.Thread`;
- официальная документация может дать вам дополнительные примеры и теоретическую информацию по этой теме; ее можно найти по адресу http://boost.org/libs/thread.

Блокировка «Несколько читателей – один писатель»

Представьте, что мы разрабатываем некий онлайн-сервис. У нас есть ассоциативный контейнер зарегистрированных пользователей с информацией по ним. Этот контейнер доступен многим потокам, и он очень редко изменяется. Все операции с приведенным ниже набором выполнены потокобезопасным способом:

```
#include <unordered_map>
#include <boost/thread/mutex.hpp>
#include <boost/thread/locks.hpp>

struct user_info {
    std::string address;
    unsigned short age;

    // Другие параметры
    // ...
};

class users_online {
    typedef boost::mutex mutex_t;

    mutable mutex_t users_mutex_;
    std::unordered_map<std::string, user_info> users_;

public:
    bool is_online(const std::string& username) const {
       boost::lock_guard<mutex_t> lock(users_mutex_);
       return users_.find(username) != users_.end();
    }

    std::string get_address(const std::string& username) const {
        boost::lock_guard<mutex_t> lock(users_mutex_);
        return users_.at(username).address;
    }

    void set_online(const std::string& username, user_info&& data) {
       boost::lock_guard<mutex_t> lock(users_mutex_);
       users_.emplace(username, std::move(data));
    }

    // Другие методы:
    // ...
};
```

К сожалению, наш онлайн-сервис работает медленно, и профилировщики показывают, что проблема кроется в классе `users_online`. Любая операция захватывает `mutex_`, поэтому даже получение информации приводит к ожиданию на мьютексе. Поскольку некоторые пользовательские данные долго копировать, мы проводим много времени в критических секциях, замедляя все операции над классом `users_online`.

К сожалению, требования проекта не позволяют нам переделать класс. Можно ли ускорить его работу без изменения интерфейса?

Подготовка

Убедитесь, что вы не испытываете неудобства при работе с `boost::thread` или `std::thread` и знаете основы мьютексов.

Как это делается...

Возможно, вам поможет следующее.

Замените boost::mutex на boost::shared_mutex. Замените boost::unique_locks на boost::shared_lock для методов, которые не изменяют данные:

```
#include <boost/thread/shared_mutex.hpp>

class users_online {
    typedef boost::shared_mutex mutex_t;

    mutable mutex_t                             users_mutex_;
    std::unordered_map<std::string, user_info>  users_;

public:
    bool is_online(const std::string& username) const {
        boost::shared_lock<mutex_t> lock(users_mutex_);
        return users_.find(username) != users_.end();
    }

    std::string get_address(const std::string& username) const {
        boost::shared_guard<mutex_t> lock(users_mutex_);
        return users_.at(username).address;
    }

    void set_online(const std::string& username, user_info&& data) {
        boost::lock_guard<mutex_t> lock(users_mutex_);
        users_.emplace(username, std::move(data));
    }

    // Другие методы:
    // ...
};
```

Как это работает...

Мы можем разрешить читать данные из нескольких потоков одновременно. Нам необходимо уникально владеть мьютексом, только если мы собираемся модифицировать защищенные им данные. И именно для таких случаев был разработан boost::shared_mutex. Он допускает совместный захват мьютекса на чтение, что позволяет множественный одновременный доступ к ресурсу.

Когда мы пытаемся захватить уникальный доступ к ресурсу, который уже захвачен на чтение, операция будет ожидать до тех пор, пока не останется никаких читателей. После того как мьютекс будет захвачен на запись, новые читатели будут ожидать его освобождения.

Захват boost::shared_mutex для чтения или записи происходит намного медленнее, чем захват обычного boost::mutex. Используйте boost::shared_mutex, только если вы уверены, что нет другого способа ускорить ваш код и бенчмарки показывают, что производительность действительно улучшилась.

Некоторые читатели, возможно, впервые видят ключевое слово `mutable`. Это слово может применяться к нестатическим и неконстантным членам класса. Член данных `mutable` можно изменить в константных функциях-членах, и обычно он используется для мьютексов и других вспомогательных переменных, которые не имеют прямого отношения к логике класса.

Дополнительно...

Когда вам нужны только уникальные доступы к ресурсам, не используйте `boost::shared_mutex`, потому что он медленнее, чем обычный `boost::mutex`.

Совместно используемые мьютексы не были доступны в C++ до появления стандарта C++14. `shared_timed_mutex` и `shared_lock` определены в заголовке `<shared_mutex>` в пространстве имен `std::`. Они имеют характеристики производительности, близкие к версиям Boost, поэтому учитывайте предыдущие замечания по производительности.

В C++17 появился `shared_mutex`, который может быть немного быстрее, чем `shared_timed_mutex`, за счет того, что он не обеспечивает средств для освобождения по времени. Это может сэкономить вам несколько драгоценных наносекунд.

См. также

- Существует также класс `boost::upgrade_mutex`, который может быть полезен в тех случаях, когда совместный захват требует перехода к уникальному захвату. См. документацию по библиотеке `Boost.Thread` по адресу http://boost.org/libs/thread для получения дополнительной информации;
- перейдите по ссылке http://herbsutter.com/2013/01/01/video-you-dont-know-const-and-mutable/ для получения дополнительной информации о ключевом слове `mutable`.

Создание переменных, уникальных для каждого потока

Давайте взглянем на рецепт создания класса `work_queue`. Каждую задачу можно выполнить в одном из множества потоков, и мы не знаем, в каком. Представьте, что мы хотим отправить результаты выполненного задания по какому-либо сетевому соединению:

```
#include <boost/noncopyable.hpp>

class connection: boost::noncopyable {
public:
    // Открытие соединения - медленная операция
    void open();

    void send_result(int result);

    // Другие методы
    // ...
};
```

У нас есть следующие решения для работы с соединениями:

- открыть новое соединение, когда нам нужно отправить данные (что очень медленно);
- использовать одно соединение для всех потоков, доступ к соединению разграничивать через мьютекс (что тоже медленно);
- держать пул соединений, получать соединение из него потокобезопасным способом (требуется много кода, но это хорошее производительное решение);
- иметь одно соединение на поток (это быстро работает и просто реализовать).

Итак, как же реализовать последнее решение?

Подготовка

Требуются базовые знания потоков.

Как это делается...

Настало время создать локальную переменную потока (англ. **thread local variable**). Объявите функцию в заголовочном файле после определения класса `connection`:

```
connection& get_connection();
```

Сделайте, чтобы ваш файл с имплементацией выглядел так:

```
#include <boost/thread/tss.hpp>
boost::thread_specific_ptr<connection> connection_ptr;

connection& get_connection() {
    connection* p = connection_ptr.get();
    if (!p) {
        connection_ptr.reset(new connection);
        p = connection_ptr.get();
        p->open();
    }

    return *p;
}
```

Готово. Использовать ресурс, специфичный для потока, никогда не было так просто:

```
void task() {
    int result;
    // Здесь выполняются какие-то вычисления.
    // ...

    // Отправка результата:
    get_connection().send_result(result);
}
```

Как это работает...

Переменная boost::thread_specific_ptr содержит отдельный указатель для каждого потока.

Первоначально этот указатель равен nullptr; вот почему мы выполняем проверку для !p и открываем соединение, если оно равно nullptr.

Итак, когда мы входим в функцию get_connection() из потока, который уже инициировал указатель, !p возвращает значение false, и мы возвращаем уже открытое соединение.

Для указателя, хранящегося в переменной connection_ptr, вызывается delete при выходе из потока выполнения, поэтому нам не нужно беспокоиться об утечках памяти.

Дополнительно...

Вы можете предоставить собственную функцию очистки, которая будет вызываться вместо delete при выходе из потока выполнения. Функция очистки должна иметь сигнатуру void (*cleanup_function)(T*) и передаваться во время создания boost::thread_specific_ptr.

В C++11 имеется специальное ключевое слово thread_local для объявления локальных переменных потока. В C++11 нет класса thread_specific_ptr, но вы можете использовать thread_local T или thread_local std::unique_ptr<T>, чтобы добиться того же поведения на компиляторах, которые поддерживают thread_local. В отличие от thread_local, класс boost::thread_specific_ptr работает и на старых компиляторах , не поддерживающих C++11.

В C++17 есть переменные inline, и вы можете использовать thread_local с этими переменными для объявления локальных переменных потока в заголовочных файлах.

См. также

- Документация по библиотеке Boost.Thread дает много хороших примеров для разных случаев; ее можно найти по адресу http://boost.org/libs/thread;
- прочитав https://stackoverflow.com/questions/13106049/what-is-the-performance-penalty-of-c11-thread-local-variables-in-gcc-4-8 и о ключевом слове __thread на странице http://gcc.gnu.org/onlinedocs/gcc-3.3.1/gcc/Thread-Local.html, вы можете составить представление по поводу того, как thread_local реализуется в компиляторах и насколько он быстр.

Прерывание потока

Иногда нужно остановить работу потока, который съел слишком много ресурсов или просто выполняется слишком долго. Например, какой-то парсер работает в потоке (и активно использует Boost.Thread), но мы уже получили необходимое количество данных, поэтому парсинг можно остановить. Вот заготовка подобного кода:

```
int main() {
    boost::thread parser_thread(&do_parse);
```

```
    // ...

    if (stop_parsing) {
        // Парсинг больше не требуется.
        // TODO: Остановить поток парсера!
    }

    // ...

    parser_thread.join();
}
```

Как же остановить парсер?

Подготовка

Для этого рецепта почти ничего не требуется. Только базовые знания потоков.

Как это делается...

Можно остановить поток, прервав его:

```
if (stop_parsing) {
    // Парсинг больше не требуется.
    parser_thread.interrupt();
}
```

Как это работает...

Библиотека `Boost.Thread` предоставляет предопределенные **точки прерывания**, в которых поток проверяется на прерывание с помощью вызова функции `interrupt()`. Если поток выполнения был прерван, выбрасывается исключение `boost::thread_interrupted`. Исключение, пролетая через внутренности функции `do_parse()`, вызывает деструкторы для всех ресурсов, как это делает любое исключение. Исключения `boost::thread_interrupted` по-особому обрабатываются библиотекой `Boost.Thread`, им разрешается выходить из функции потока (в нашем примере это `do_parse()`). Когда исключение покидает функцию потока выполнения, оно перехватывается внутренними механизмами `boost::thread` и трактуется как запрос на отмену потока.

`boost::thread_interrupted` не наследуется от `std::exception`! Прерывания работают хорошо, если вы перехватываете исключения по их типу или по ссылкам на `std::exception`. Но если вы перехватили исключение с помощью `catch(...)` и не выбрасываете перехваченное исключение, прерывания не сработают.

Как мы знаем из первого рецепта этой главы, если функция, переданная в поток, не перехватывает исключение, а исключение выходит за пределы функции, приложение завершает работу. `boost::thread_interrupted` является единственным исключением из этого правила; он может выходить за пределы функций и не вызывать `std::terminate()`.

Дополнительно...

Точки прерывания библиотеки Boost.Thread перечислены в официальной документации. Как правило, все, что потенциально блокирует поток выполнения, проверяет на прерывания.

Также можно вручную добавить точки прерывания в любом месте. Все, что нам нужно, – это вызвать функцию boost::this_thread::interruptio_point():

```
void do_parse() {
    while (not_end_of_parsing) {
        // Если текущий поток был прерван, приведенная ниже
        // строка выбросит исключение boost::thread_interrupted.
        boost::this_thread::interruption_point();
        // Здесь идет парсинг
        // ...
    }
}
```

Если для проекта прерывания не требуются, выставление макроса BOOST_THREAD_DONT_PROVIDE_INTERRUPTIONS дает небольшое повышение производительности и полностью отключает прерывания потоков.

В C++11 нет прерываний для потоков выполнения, но вы можете частично эмулировать их, используя атомарные операции:

- создайте атомарную переменную bool;
- проверьте атомарную переменную в потоке и сгенерируйте исключение, если она изменилась;
- не забудьте перехватить это исключение в функции, передаваемой потоку (в противном случае ваше приложение завершит работу).

Однако это не поможет, если код ждет где-то в мьютексе, условной переменной или в методе sleep.

См. также

- Официальная документация по Boost.Thread содержит список предопределенных точек прерывания http://www.boost.org/doc/libs/1_64_0/doc/html/thread/thread_management.html#thread.thread_management.tutorial.interrupt.predefined_interruption_points;
- в качестве упражнения посмотрите другие рецепты из этой главы и подумайте, где дополнительные точки прерывания улучшат код;
- чтение других разделов документации по Boost.Thread может быть полезно; перейдите по ссылке http://boost.org/libs/thread.

Манипулирование группой потоков

Тем читателям, которые экспериментировали с потоками и пытались повторить все примеры самостоятельно, уже надоело писать этот код для запуска потоков и выполнения функции-члена join:

```
#include <boost/thread.hpp>

void some_function();
```

```
void sample() {
    boost::thread t1(&some_function);
    boost::thread t2(&some_function);
    boost::thread t3(&some_function);

    // ...

    t1.join();
    t2.join();
    t3.join();
}
```

Может быть, есть способ получше?

Подготовка

Базовых знаний потоков будет более чем достаточно для этого рецепта.

Как это делается...

Мы можем манипулировать группой потоков, используя класс `boost::thread_group`.

1. Создайте переменную `boost::thread_group`:

```
#include <boost/thread.hpp>

int main() {
    boost::thread_group threads;
```

2. Создайте потоки в предыдущей переменной:

```
// Запуск 10 потоков
for (unsigned i = 0; i < 10; ++i) {
    threads.create_thread(&some_function);
}
```

3. Теперь вы можете вызывать функции для всех потоков внутри `boost::thread_group`:

```
    // Выполнение функции join для всех потоков.
    threads.join_all();

    // Мы также можем прервать их все,
    // вызвав функцию threads.interrupt_all();
}
```

Как это работает...

Переменная `boost::thread_group` просто содержит все потоки, добавленные в нее, и может отправлять вызовы всем потокам.

Дополнительно...

В C++11 нет класса `thread_group`; этот класс есть только в Boost.

См. также

Официальная документация по Boost.Thread может удивить вас множеством других полезных классов, которые не описаны в этой главе; перейдите на страницу http://boost.org/libs/thread.

Безопасная инициализация общей переменной

Представьте, что мы разрабатываем критически важный для безопасности класс, который используется из нескольких потоков, получает ответы от сервера, обрабатывает их и выводит ответ:

```
struct postprocessor {
    typedef std::vector<std::string> answer_t;

    // Параллельные вызовы для одного и того же postprocessor являются безопасными.
    answer_t act(const std::string& in) const {
        if (in.empty()) {
            // Крайне редкое условие.
            return read_defaults();
         }

         // ...
    }
};
```

Обратите внимание на строку `return read_defaults();`. Бывают ситуации, когда сервер не отвечает из-за проблем, связанных с сетью или еще чего-то. В этих случаях мы пытаемся прочитать значения по умолчанию из файла и ответить заранее заготовленным ответом:

```
// Выполняется долго.
std::vector<std::string> read_defaults();
```

Из предыдущего кода видно, что мы столкнулись с проблемой: сервер может быть недоступен в течение некоего заметного периода времени, и все это время мы будем перечитывать файл при каждом вызове функции `act`. Это существенно влияет на производительность.

Можно попытаться исправить это, сохранив заранее заготовленный ответ `default_` внутри класса:

```
struct postprocessor {
    typedef std::vector<std::string> answer_t;

private:
    answer_t default_;

public:
    postprocessor()
        : default_(read_defaults())
    {}

    // Параллельные вызовы для одной и той же переменной являются безопасными.
    answer_t act(const std::string& in) const {
```

```
        if (in.empty()) {
            // Крайне редкое условие
            return default_;
        }

        // ...
    }
};
```

Это также не идеальное решение: мы не знаем, сколько экземпляров класса `postprocessor` создается пользователем, и мы тратим память на значения по умолчанию, которые могут не потребоваться во время выполнения.

Таким образом, мы должны одновременно безопасно читать и сохранять данные в текущем экземпляре класса при первом сбое удаленного сервера, и не перечитывать данные при следующих сбоях. Есть много способов сделать это, но давайте рассмотрим самый правильный.

Подготовка

Базовых знаний потоков более чем достаточно для этого рецепта.

Как это делается...

1. Мы должны добавить переменные для хранения информации о том, что значение было прочитано, и переменную для хранения самого значения:

```
#include <boost/thread/once.hpp>

struct postprocessor {
    typedef std::vector<std::string> answer_t;

private:
    mutable boost::once_flag default_flag_;
    mutable answer_t default_;
```

Переменные являются изменяемыми (`mutable`), потому что мы собираемся изменить их внутри функций-членов `const`.

2. Давайте проинициализируем наши переменные:

```
public:
    postprocessor()
        : default_flag_(BOOST_ONCE_INIT)
        , default_()
     {}
```

3. И наконец, изменим функцию `act`:

```
// Параллельные вызовы для одной и той же переменной являются безопасными.
answer_t act(const std::string& in) const {
    answer_t ret;
    if (in.empty()) {
        // Крайне редкое условие.
        boost::call_once(default_flag_, [this]() {
            this->default_ = read_defaults();
        });
```

```
            return default_;
        }

        // ...
        return ret;
    }
};
```

Как это работает...

Говоря кратко, для одной и той же переменной `boost::once_flag` второй аргумент функции `boost::call_once` вызывается только один раз. Если одновременно происходит два или более параллельных вызова для одного и того же `once_flag`, то функция все равно выполнится только *один раз*.

Если при вызове функции не было выброшено исключений, которые вышли за пределы ее тела, тогда `boost::call_once` предполагает, что вызов был успешным, и сохраняет эту информацию в `boost::once_flag`. Любые последующие вызовы функции `boost::call_once` с тем же `boost::once_flag` ничего не делают.

Не забывайте инициализировать функцию `boost::once_flag` с помощью макроса `BOOST_ONCE_INIT`.

Дополнительно...

`boost::call_once` может передавать параметры в функцию для вызова:

```
#include <iostream>

void once_printer(int i) {
    static boost::once_flag flag = BOOST_ONCE_INIT;
    boost::call_once(
        flag,
        [](int v) { std::cout << "Print once " << v << '\n'; },
        i // <=== Передается в лямбда-функцию, которая идет выше.
    );

    // ...
}
```

Теперь если мы вызовем функцию `once_printer` в цикле:

```
int main() {
    for (unsigned i = 0; i < 10; ++i) {
        once_printer(i);
    }
}
```

на выходе будет только одна строка:

```
Print once 0
```

В C++11 есть `std::call_once` и `std::once_flag` в заголовке `<mutex>`. В отличие от версии Boost, версия `once_flag` стандартной библиотеки не требует инициа-

лизации через макрос, у нее есть constexpr-конструктор. Как обычно, версия Boost может использоваться на компиляторах для стандартов, предшествующих C++11, поэтому применяйте ее, если вам нужно обеспечить поддержку старых компиляторов.

До 2015 года Visual Studio поставляла неоптимальную реализацию `std::call_once` более чем в десять раз медленнее, чем версия Boost. Придерживайтесь `boost::call_once`, если вы не используете современные компиляторы.

См. также

Документация по `Boost.Thread` дает множество хороших примеров для разных случаев. Ее можно найти по адресу http://boost.org/libs/thread.

Захват нескольких мьютексов

В следующих нескольких параграфах вы станете одним из тех, кто пишет игры. Поздравляем, вы можете играть на работе!

Вы разрабатываете сервер, и вам нужно написать код для обмена добычей между двумя пользователями:

```
class user {
    boost::mutex loot_mutex_;
    std::vector<item_t> loot_;
public:
    // ...

    void exchange_loot(user& u);
};
```

Каждое действие пользователя может одновременно обрабатываться разными потоками на сервере, поэтому вы должны защищать ресурсы мьютексами. Младший разработчик попытался разобраться с проблемой, но его решение не работает:

```
void user::exchange_loot(user& u) {
  // Совсем неправильно!!! Взаимоблокировка типа ABBA.
  boost::lock_guard<boost::mutex> l0(loot_mutex_);
  boost::lock_guard<boost::mutex> l1(u.loot_mutex_);
  loot_.swap(u.loot_);
}
```

Проблема предыдущего кода – хорошо известная **взаимоблокировка типа ABBA**. Представьте, что *поток 1* захватывает *мьютекс A*, а *поток 2* – *мьютекс B*. И теперь **поток 1** пытается захватить уже захваченный *мьютекс B*, а *поток 2* пытается захватить уже захваченный *мьютекс A*. Это приводит к тому, что два потока бесконечно блокируются друг другом, поскольку им нужен ресурс, захваченный другим потоком.

Теперь, если `user1` и `user2` параллельно вызывают `exchange_loot` друг для друга, мы можем столкнуться с ситуацией, когда `user1.exchange_loot(user2)` захваты-

вает `user1.loot_mutex_`, а `user2.exchange_loot(user1)` захватывает `user2.loot_mutex_`. `user1.exchange_loot(user2)` до бесконечности ожидает `user2.loot_mutex_`, а `user2.exchange_loot(user1)` бесконечно ожидает `user1.loot_mutex_`.

Подготовка

Для этого рецепта достаточно базовых знаний потоков и мьютексов.

Как это делается...

Существует два основных готовых решения этой проблемы.

1. Короткое решение, которое требует от компилятора поддержки вариативного шаблона:

```
#include <boost/thread/lock_factories.hpp>

void user::exchange_loot(user& u) {
    typedef boost::unique_lock<boost::mutex> lock_t;

    std::tuple<lock_t, lock_t> l = boost::make_unique_locks(
        loot_mutex_, u.loot_mutex_
    );

    loot_.swap(u.loot_);
}
```

Тот же код с использованием `auto`:

```
#include <boost/thread/lock_factories.hpp>

void user::exchange_loot(user& u) {
    auto l = boost::make_unique_locks(
        loot_mutex_, u.loot_mutex_
    );

    loot_.swap(u.loot_);
}
```

2. Переносимое решение:

```
#include <boost/thread/locks.hpp>

void user::exchange_loot(user& u) {
    typedef boost::unique_lock<boost::mutex> lock_t;

    lock_t l0(loot_mutex_, boost::defer_lock);
    lock_t l1(u.loot_mutex_, boost::defer_lock);
    boost::lock(l0, l1);

    loot_.swap(u.loot_);
}
```

Как это работает...

Основная идея заключается в том, чтобы как-то упорядочить мьютексы и всегда захватывать их, следуя этому порядку. В этом случае проблема взаимобло-

кировки типа ABBA невозможна, поскольку все потоки всегда захватывают мьютекс *A* до мьютекса *B*. Обычно используются другие алгоритмы предотвращения взаимоблокировок, но для простоты приведенного здесь примера мы предполагаем, что используется упорядочение мьютексов.

В первом примере мы использовали функцию `boost::make_unique_locks`, которая всегда захватывает мьютексы в определенном порядке и возвращает кортеж, содержащий `boost::unique_lock`.

Во втором примере мы создали переменные `boost::unique_lock` вручную, но не захватили ими мьютекс благодаря переданному параметру `boost::defer_lock`. Фактический захват мьютексом произошел в вызове `boost::lock(l0, l1)`, который заблокировал мьютексы в предопределенном порядке.

Теперь, если `user1` и `user2` одновременно вызывают `exchange_loot` друг для друга, вызовы `user1.exchange_loot(user2)` и `user2.exchange_loot(user1)` сначала попытаются заблокировать `user1.loot_mutex_`, либо оба попытаются сначала заблокировать `user2.loot_mutex_`.

Дополнительно…

Функции `boost::make_unique_locks` и `boost::lock` могут принимать более двух мьютексов, поэтому вы можете использовать их в более сложных случаях, когда нужно одновременно захватить более двух мьютексов.

В C++11 есть функция `std::lock`, определенная в заголовке `<mutex>`, которая ведет себя точно так же, как функция `boost::lock`.

C++17 предлагает гораздо более красивое решение:

```
#include <mutex>
void user::exchange_loot(user& u) {
    std::scoped_lock l(loot_mutex_, u.loot_mutex_);
    loot_.swap(u.loot_);
}
```

В предыдущем коде `std::scoped_lock` – это класс, который принимает различное количество мьютекстов. Он имеет параметры вариативного шаблона, которые автоматически выводятся из руководства по выводу (англ. deduction guides) C++17. Фактический тип класса `std::scoped_lock` из предыдущего примера:

```
std::scoped_lock<std::mutex, std::mutex>
```

`std::scoped_lock` захватывает все мьютексы, переданные во время конструирования, и позволяет избежать взаимных блокировок. Другими словами, это работает как первый пример, но выглядит немного лучше.

См. также

Официальная документация по `Boost.Thread` может удивить вас множеством других полезных классов, которые не были описаны в этой главе; перейдите по ссылке http://boost.org/libs/thread.

Глава 6

Манипулирование задачами

Темы, которые мы рассмотрим в этой главе:

- регистрация задачи для обработки произвольного типа данных;
- создание таймеров и обработка событий таймера в качестве задач;
- передача данных по сети в качестве задачи;
- прием входящих соединений;
- параллельное выполнение различных задач;
- конвейерная обработка задач;
- создание неблокирующего барьера;
- хранение исключения и создание из него задачи;
- получение и обработка системных сигналов в виде задач.

Вступление

Эта глава посвящена задачам. Мы будем называть функциональный объект *задачей*, потому что так короче и это лучше отражает то, что он будет делать. Основная идея данной главы заключается в том, что мы можем разделить всю обработку, вычисления и взаимодействия на функторы (задачи) и обрабатывать каждую из этих задач практически независимо. Более того, мы можем не ждать завершения некоторых медленных операций, таких как получение данных из сокета или ожидание тайм-аута, а вместо этого предоставить задачу обратного вызова и продолжить работу с другими задачами. Как только ОС завершает выполнение медленной операции, она вызывает нашу задачу.

Лучший способ понять пример – поэкспериментировать с ним, изменяя, запуская и расширяя его. На сайте http://apolukhin.github.io/Boost-Cookbook/ есть все примеры из этой главы, и вы даже можете поэкспериментировать с некоторыми из них онлайн.

Прежде чем вы начнете

Эта глава требует по крайней мере базовых знаний первой, второй и пятой глав. Необходимы базовые знания по rvalue-ссылкам C++11 и лямбда-выражениям.

Регистрация задачи для обработки произвольного типа данных

Прежде всего давайте позаботимся о классе, который содержит все задачи и предоставляет методы для их выполнения. Мы уже делали нечто подобное в рецепте «*Создание класса work_queue*» главы 5 «*Многопоточность*», но некоторые из приведенных ниже проблем не были решены:

- класс `work_queue` только хранит и возвращает задачи, но нам также необходимо выполнять существующие задачи;
- задача может выбросить исключение. Нам нужно перехватывать и обрабатывать исключения, если они выходят за пределы задач;
- задача может не заметить прерывание потока. Следующая задача в том же потоке может получить прерывание, которое предназначалось не ей;
- нам нужен способ остановить обработку задач.

Подготовка

Этот рецепт требует линковки с библиотеками `boost_system` и `boost_thread`, а также необходимы базовые знания `Boost.Thread`.

Как это делается...

В этом рецепте мы используем `boost::asio::io_service` вместо `work_queue` из предыдущей главы. Для этого есть причина, и мы увидим ее в последующих рецептах.

1. Давайте начнем со структуры, которая оборачивает пользовательскую задачу:

```
#include <boost/thread/thread.hpp>
#include <iostream>

namespace detail {
template <class T>
struct task_wrapped {
private:
    T task_unwrapped_;

public:
    explicit task_wrapped(const T& f)
        : task_unwrapped_(f)
    {}

void operator()() const {
    // Сброс прерывания.
    try {
        boost::this_thread::interruption_point();
    } catch(const boost::thread_interrupted&){}

    try {
         // Выполнение задачи.
         task_unwrapped_();
    } catch (const std::exception& e) {
```

```
            std::cerr<< "Exception: " << e.what() << '\n';
        } catch (const boost::thread_interrupted&) {
            std::cerr<< "Thread interrupted\n";
        } catch (...) {
            std::cerr<< "Unknown exception\n";
        }
    }
};

} // namespace detail
```

2. Для простоты использования мы добавим функцию, которая создает task_wrapped из функтора пользователя:

```
namespace detail {

template <class T>
task_wrapped<T> make_task_wrapped(const T& task_unwrapped) {
    return task_wrapped<T>(task_unwrapped);
}

} // namespace detail
```

3. Теперь мы готовы написать класс tasks_processor:

```
#include <boost/asio/io_service.hpp>

class tasks_processor: private boost::noncopyable {
protected:
    static boost::asio::io_service& get_ios() {
        static boost::asio::io_service ios;
        static boost::asio::io_service::work work(ios);

        return ios;
    }
```

4. Давайте добавим метод push_task:

```
public:
    template <class T>
    static void push_task(const T& task_unwrapped) {
        get_ios().post(detail::make_task_wrapped(task_unwrapped));
    }
```

5. Закончим этот класс, добавив функции-члены для запуска и остановки цикла выполнения задачи:

```
    static void start() {
        get_ios().run();
    }

    static void stop() {
        get_ios().stop();
    }
}; // tasks_processor
```

Готово! Теперь пришло время протестировать наш класс:

```
int func_test() {
    static int counter = 0;
    ++ counter;
    boost::this_thread::interruption_point();

    switch (counter) {
    case 3:
       throw std::logic_error("Just checking");

    case 10:
       // Эмуляция прерывания потока.
       // Перехват внутри task_wrapped. Выполнение следующих задач продолжится.
       throw boost::thread_interrupted();

   case 90:
       // Остановка task_processor.
       tasks_processor::stop();
   }

   return counter;
}
```

Функция main может выглядеть так:

```
int main () {
    for (std::size_t i = 0; i < 100; ++i) {
         tasks_processor::push_task(&func_test);
    }

    // Обработка не была начата.
    assert(func_test() == 1);

    // Мы также можем использовать лямбда-выражение в качестве задачи.
    // Асинхронно считаем 2 + 2.
    int sum = 0;
    tasks_processor::push_task(
        [&sum]() { sum = 2 + 2; }
    );

    // Обработка не была начата.
    assert(sum == 0);

   // Не выбрасывает исключение, но блокирует текущий поток выполнения до тех пор,
   // пока одна из задач не вызовет tasks_processor::stop().
   tasks_processor::start();
   assert(func_test() == 91);
}
```

Как это работает...

Переменная boost::asio::io_service может хранить и выполнять опубликованные в ней задачи. Но мы не можем публиковать задачи пользователя на-

прямую, потому что они могут получить прерывание, адресованное другим задачам, или выбросить исключение. Вот почему мы оборачиваем задачу пользователя в структуру `detail::task_wrapped`. Она сбрасывает все предыдущие прерывания, вызывая:

```
try {
    boost::this_thread::interruption_point();
} catch(const boost::thread_interrupted&){}
```

`detail::task_wrapped` выполняет задачу в блоке `try{} catch()`, следя за тем, чтобы исключение не выходило за пределы `operator()`.

Посмотрите на функцию `start()`. Метод `boost::asio::io_service::run()` начинает обработку задач, размещенных в переменной `io_service`. Если `boost::asio::io_service::run()` не вызывается, то опубликованные задачи не выполняются (это можно увидеть в функции `main()`). Обработку задачи можно остановить с помощью вызова `boost::asio::io_service::stop()`.

Класс `boost::asio::io_service` выходит из функции `run()`, если задач больше не осталось, поэтому мы заставляем его продолжать выполнение с помощью экземпляра `boost::asio::io_service::work`:

```
static boost::asio::io_service& get_ios() {
    static boost::asio::io_service ios;
    static boost::asio::io_service::work work(ios);

    return ios;
}
```

Переменные из `<iostream>`, такие как `std::cerr` и `std::cout`, не являются потокобезопасными в компиляторах до C++11 и могут перемежать данные из разных потоков в компиляторах, совместимых с C++11. В реальных проектах должна использоваться дополнительная синхронизация, чтобы получить читаемый вывод. Для простоты примера мы этого не делаем.

Дополнительно...

В стандартной библиотеке C++20 и более ранних версиях нет `io_service`. Тем не менее большая часть библиотеки `Boost.Asio` предлагается в качестве Networking TS как дополнение к C++.

См. также

- Приведенные далее рецепты в этой главе покажут вам, почему мы выбираем `boost::asio::io_service`, вместо того чтобы использовать наш рукописный код из главы 5 «*Многопоточность*»;
- вы можете ознакомиться с документацией по `Boost.Asio` для получения примеров, учебных пособий и ссылок на классы по адресу http://boost.org/libs/asio;
- вы также можете прочитать книгу *Boost.Asio C++ Network Programming*, в которой дается более плавное введение в `Boost.Asio` и которая охватывает некоторые детали, которые не были освещены в этой книге.

Создание таймеров и обработка событий таймера в качестве задач

Проверка чего-либо с заданными интервалами – распространенная задача. Например, нам нужно проверять сессии на активность раз в 5 секунд. Есть популярные решения для такой проблемы:

- мы создаем поток, который выполняет проверку, а затем спит в течение 5 секунд. Это слабое решение, которое потребляет много системных ресурсов и плохо масштабируется;
- при правильном решении используются системные API для управления таймерами асинхронно. Это более подходящее решение. С ним придется потрудиться, и вы получите непереносимый код, если не воспользуетесь библиотекой `Boost.Asio`.

Подготовка

Вы должны знать, как использовать rvalue-ссылки и `unique_ptr`. Этот рецепт основан на коде из предыдущего рецепта. *См.* первый рецепт этой главы, чтобы получить информацию о классах `boost::asio::io_service` и `task_queue`.

Линкуйте пример с библиотеками `boost_system` и `boost_thread`. Определите `BOOST_ASIO_DISABLE_HANDLER_TYPE_REQUIREMENTS`, чтобы обойти устаревшие проверки библиотеки `Boost.Asio`.

Как это делается...

Мы просто модифицируем класс `tasks_processor`, добавляя новые методы для запуска задачи в определенное время.

1. Давайте добавим метод в наш класс `tasks_processor` для отложенного запуска задачи:

```
class tasks_processor {
    // ...
public:
    template <class Time, class Func>
    static void run_delayed(Time duration_or_time, const Func& f) {
        std::unique_ptr<boost::asio::deadline_timer> timer(
            new boost::asio::deadline_timer(
                get_ios(), duration_or_time
            )
        );

        timer_ref.async_wait(
            detail::timer_task<Func>(
                std::move(timer),
                f
            )
        );
    }
};
```

2. В качестве последнего шага мы создаем структуру `timer_task`:

```
#include <boost/asio/io_service.hpp>
#include <boost/asio/deadline_timer.hpp>
#include <boost/system/error_code.hpp>
#include <memory> // std::unique_ptr
#include <iostream>

namespace detail {

    template <class Functor>
    struct timer_task {
    private:
        std::unique_ptr<boost::asio::deadline_timer> timer_;
        task_wrapped<Functor> task_;

public:
    explicit timer_task(
            std::unique_ptr<boost::asio::deadline_timer> timer,
            const Functor& task_unwrapped)
        : timer_(std::move(timer))
        , task_(task_unwrapped)
   {}

   void operator()(const boost::system::error_code& error) const {
       if (!error) {
           task_();
       } else {
           std::cerr << error << '\n';
       }
    }
  };
} // namespace detail
```

Вот как можно было бы использовать новую функциональность:

```
int main () {
    const int seconds_to_wait = 3;
    int i = 0;

    tasks_processor::run_delayed(
        boost::posix_time::seconds(seconds_to_wait),
        test_functor(i)
    );

    tasks_processor::run_delayed(
        boost::posix_time::from_time_t(time(NULL) + 1),
        &test_func1
    );

    assert(i == 0);

   // Блокирует, пока одна из задач не вызовет tasks_processor::stop().
   tasks_processor::start();
}
```

где test_functor – это структура с определенным operator(), а test_func1 – функция:

```
struct test_functor {
    int& i_;
    explicit test_functor(int& i);
    void operator()() const {
        i_ = 1;
        tasks_processor::stop();
    }
};

void test_func1();
```

Как это работает...

Говоря кратко, по истечении заданного промежутка времени boost::asio::deadline_timer помещает задачу в экземпляр класса boost::asio::io_service для выполнения.

Все неприятные вещи находятся внутри функции run_delayed:

```
template <class Time, class Functor>
static void run_delayed(Time duration_or_time, const Functor& f) {
    std::unique_ptr<boost::asio::deadline_timer>
    timer( /* ... */ );

    boost::asio::deadline_timer& timer_ref = *timer;

    timer_ref.async_wait(
        detail::timer_task<Functor>(
            std::move(timer),
            f
        )
    );
}
```

Функция tasks_processor::run_delayed принимает тайм-аут и функтор для вызова после тайм-аута. В ней создается уникальный указатель на boost::asio::deadline_timer. boost::asio::deadline_timer содержит специфичные для платформы вещи для асинхронного выполнения задачи по таймеру.

Boost.Asio не управляет памятью «из коробки». Пользователь библиотеки должен сам позаботиться об управлении ресурсами, обычно располагая их в задаче. Так что если нам нужен таймер и мы хотим, чтобы какая-то функция выполнялась после указанного тайм-аута, мы должны переместить уникальный указатель таймера в задачу, получить ссылку на него и передать полученную задачу таймеру.

В этой строке мы получаем ссылку на deadline_timer:

```
boost::asio::deadline_timer& timer_ref = *timer;
```

Теперь создаем объект detail::timer_task, который хранит функтор и получает во владение unique_ptr<boost::asio::deadline_timer>:

```
detail::timer_task<Functor>(
    std::move(timer),
    f
)
```

boost::asio::deadline_timer нельзя уничтожать до тех пор, пока он не сработает, и перемещение его в функтор timer_task гарантирует это.

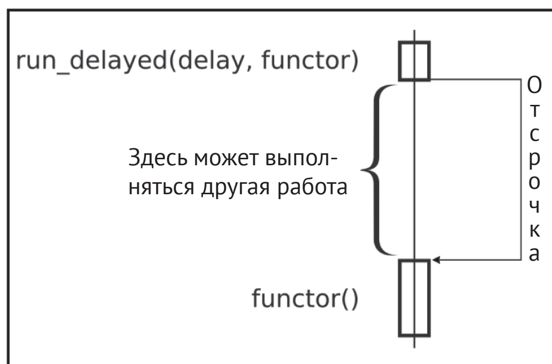
Наконец, мы инструктируем boost::asio::deadline_timer отправить функтор timer_task в io_service по истечении запрошенного промежутка времени:

```
timer_ref.async_wait( /* timer_task */ )
```

Ссылка на переменную io_service хранится внутри переменной boost::asio::deadline_timer. Вот почему его конструктору требуется ссылка на io_service, чтобы сохранить ее и использовать для публикации задачи по истечении тайм-аута.

Метод detail::timer_task::operator() принимает boost::syste::error_code, который будет содержать описание ошибки, если во время ожидания произошло что-то плохое. Если ошибка не произошла, мы вызываем функтор пользователя, который обернут для перехвата исключений (мы повторно используем структуру detail::task_wrapped из первого рецепта).

boost::asio::deadline_timer::async_wait не потребляет ресурсы ЦП и не блокирует поток выполнения при ожидании тайм-аута. Вы можете добавлять задачи с тайм-аутами в io_service, и они начнут выполняться только после тог, о как ОС выждет заданные интервалы времени.



Возьмите себе за правило: все ресурсы, которые используются во время вызовов async_*, должны храниться в задании.

Дополнительно...

Некоторые экзотические и устаревшие платформы не имеют API для подходящей реализации таймеров, поэтому библиотека Boost.Asio эмулирует поведение асинхронного таймера, используя дополнительный поток выполнения для io_service. Для таких платформ нет другого способа эффективно реализовать таймеры.

В C++ нет Boost.Asio-подобных классов; однако в Networking TS есть классы async_wait и timer.

См. также

- Прочитав первый рецепт из этой главы, вы научитесь основам boost::asio::io_service. Приведенные далее рецепты предоставят вам больше примеров использования io_service и покажут, как работать с передачей данных по сети, сигналами и другими функциями, используя Boost.Asio;
- вы можете ознакомиться с документацией по Boost.Asio для получения примеров, учебных пособий и ссылок на классы на сайте http://boost.org/libs/asio.

Передача данных по сети в качестве задачи

Получение или отправка данных по сети – медленная операция. Пока пакеты принимаются машиной, а ОС проверяет их и копирует данные в указанный пользователем буфер, может пройти несколько секунд.

Вместо того чтобы ждать, мы можем сделать много полезного! Давайте изменим наш класс tasks_processor, чтобы он мог отправлять и получать данные в асинхронном режиме. Говоря простым языком, мы просим его «Получи как минимум *N* байт от удаленного хоста, а после этого вызови наш функтор. Кстати, не блокируй поток выполнения при получении!» Те читатели, которые знают о **libev**, **libevent** или Node.js, могут найти в этом рецепте много знакомых вещей.

Подготовка

Этот рецепт основан на двух предыдущих рецептах. *См.* первый рецепт этой главы, чтобы получить информацию о классах boost::asio::io_service и task_queue, и второй рецепт, где приводится обзор основ асинхронной обработки.

Линкуйте пример с библиотеками boost_system и boost_thread. Определите макрос BOOST_ASIO_DISABLE_HANDLER_TYPE_REQUIREMENTS, чтобы обойти устаревшие проверки библиотеки Boost.Asio.

Как это делается...

Давайте расширим код из предыдущего рецепта, добавив методы для создания соединений.

1. Соединение будет представлено классом connection_with_data. Этот класс хранит сокет для удаленного хоста и std::string для получения и отправки данных:

```
#include <boost/asio/ip/tcp.hpp>
#include <boost/core/noncopyable.hpp>

struct connection_with_data: boost::noncopyable {
    boost::asio::ip::tcp::socket socket;
    std::string data;
```

```
    explicit connection_with_data(boost::asio::io_service& ios)
        : socket(ios)
    {}

    void shutdown() {
        if (!socket.is_open()) {
            return;
        }

        boost::system::error_code ignore;
        socket.shutdown(
            boost::asio::ip::tcp::socket::shutdown_both,
            ignore
        );
        socket.close(ignore);
    }

    ~connection_with_data() {
        shutdown();
    }
};
```

2. Как и в предыдущем рецепте, мы будем использовать класс в основном через уникальный указатель на него. Давайте добавим для простоты псевдоним типа:

```
#include <memory> // std::unique_ptr

typedef std::unique_ptr<connection_with_data> connection_ptr;
```

3. Класс `tasks_processor` из предыдущего рецепта владеет объектом `boost::asio::io_service`. Кажется разумным превратить его в фабрику для создания соединений:

```
class tasks_processor {
    // ...
public:
    static connection_ptr create_connection(
        const char* addr,
        unsigned short port_num)
    {
        connection_ptr c( new connection_with_data(get_ios()) );
        c->socket.connect(boost::asio::ip::tcp::endpoint(
            boost::asio::ip::address_v4::from_string(addr),
            port_num
        ));

        return c;
    }
};
```

4. Ниже приведены методы асинхронной передачи данных на удаленный хост:

```
#include <boost/asio/write.hpp>

template <class T>
struct task_wrapped_with_connection;

template <class Functor>
void async_write_data(connection_ptr&& c, const Functor& f) {
    boost::asio::ip::tcp::socket& s = c->socket;
    std::string& d = c->data;

    boost::asio::async_write(
        s,
        boost::asio::buffer(d),
        task_wrapped_with_connection<Functor>(std::move(c), f)
    );
}
```

5. Далее приводятся методы асинхронного чтения данных с удаленного хоста:

```
#include <boost/asio/read.hpp>

template <class Functor>
void async_read_data(
    connection_ptr&& c,
    const Functor& f,
    std::size_t at_least_bytes)
{
    c->data.resize(at_least_bytes);
    boost::asio::ip::tcp::socket& s = c->socket;
    std::string& d = c->data;
    char* p = (d.empty() ? 0 : &d[0]);

    boost::asio::async_read(
        s,
        boost::asio::buffer(p, d.size()),
        task_wrapped_with_connection<Functor>(std::move(c), f)
    );
}

template <class Functor>
void async_read_data_at_least(
    connection_ptr&& c,
    const Functor& f,
    std::size_t at_least_bytes,
    std::size_t at_most)
{
    std::string& d = c->data;
    d.resize(at_most);
    char* p = (at_most == 0 ? 0 : &d[0]);

    boost::asio::ip::tcp::socket& s = c->socket;

    boost::asio::async_read(
        s,
```

```
            boost::asio::buffer(p, at_most),
            boost::asio::transfer_at_least(at_least_bytes),
            task_wrapped_with_connection<Functor>(std::move(c), f)
        );
    }
```

6. Последняя часть – это определение класса task_wrapped_with_connection:

```
template <class T>
struct task_wrapped_with_connection {
private:
    connection_ptr c_;
    T task_unwrapped_;

public:
    explicit task_wrapped_with_connection
   (connection_ptr&& c, const T& f)
       : c_(std::move(c))
       , task_unwrapped_(f)
   {}

    void operator()(
        const boost::system::error_code& error,
        std::size_t bytes_count)
    {
        c_->data.resize(bytes_count);
        task_unwrapped_(std::move(c_), error);
    }
};
```

Готово! Теперь пользователь библиотеки может использовать предыдущий класс для отправки данных:

```
void send_auth() {
    connection_ptr soc = tasks_processor::create_connection(
        "127.0.0.1", g_port_num
    );
    soc->data = "auth_name";
    async_write_data(
        std::move(soc),
        &on_send
    );
}
```

Пользователи также могут применять его для получения данных:

```
void receive_auth_response(
    connection_ptr&& soc,
    const boost::system::error_code& err)
{
    if (err) {
        std::cerr << "Error on sending data: "
        << err.message() << '\n';
        assert(false);
    }
```

```
    async_read_data(
        std::move(soc),
        &process_server_response,
        2
    );
}
```

Вот как пользователь библиотеки может обрабатывать полученные данные:

```
void process_server_response(
        connection_ptr&& soc,
        const boost::system::error_code& err)
{
    if (err && err != boost::asio::error::eof) {
        std::cerr << "Client error on receive: "
                  << err.message() << '\n';
        assert(false);
    }

    if (soc->data.size() != 2) {
        std::cerr << "Wrong bytes count\n";
        assert(false);
    }

    if (soc->data != "OK") {
        std::cerr << "Wrong response: " << soc->data << '\n';
        assert(false);
    }

    soc->shutdown();
    tasks_processor::stop();
}
```

Как это работает...

Библиотека Boost.Asio не управляет ресурсами и буферами по умолчанию. Таким образом, если нам нужен простой интерфейс для чтения и записи данных, самое простое решение – связать сокет и буфер для отправки/получения данных. Это то, что делает класс connection_with_data. Он содержит класс boost::asio::ip::tcp::socket, который является оберткой из Boost.Asio для нативных сокетов, и переменную std::string, которую мы используем в качестве буфера.

Конструктор класса boost::asio::ip::tcp::socket принимает boost::asio::io_service как почти все классы в Boost.Asio. После того как мы создадим сокет, он должен быть подключен к удаленной конечной точке:

```
c->socket.connect(boost::asio::ip::tcp::endpoint(
    boost::asio::ip::address_v4::from_string(addr),
    port_num
));
```

Посмотрите на функцию записи. Она принимает уникальный указатель на класс connection_with_data и функтор f:

```
#include <boost/asio/write.hpp>

template <class Functor>
void async_write_data(connection_ptr&& c, const Functor& f) {
```

В ней мы получаем ссылки на сокет и буфер:

```
    boost::asio::ip::tcp::socket& s = c->socket;
    std::string& d = c->data;
```

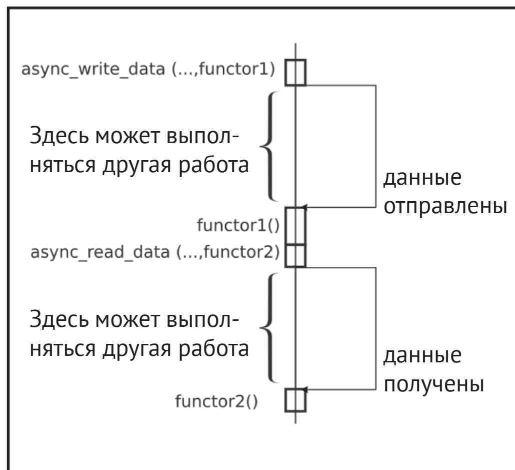
Затем просим асинхронно выполнить запись:

```
    boost::asio::async_write(
        s,
        boost::asio::buffer(d),
        task_wrapped_with_connection<Functor>(std::move(c), f)
    );
}
```

Все самое интересное происходит в функции boost::asio::async_write. Как и в случае таймеров, асинхронный вызов возвращает управление незамедлительно без выполнения функции. Он лишь дает указание, чтобы отправить задачу обратного вызова (третий параметр функции async_write) в boost::asio::io_service по завершении операции (в нашем случае это запись данных в сокет).

boost::asio::io_service выполняет нашу задачу обратного вызова в одном из потоков, который вызвал метод io_service::run(). Это показано на приведенной ниже диаграмме.

Теперь взгляните на task_wrapped_with_connection::operator(). Она принимает const boost::system::error_code& error и std::size_t bytes_count, потому что обе функции, boost::asio::async_write и boost::asio::async_read, передают эти параметры по завершении асинхронной операции. Вызов c_->data.resize(bytes_count); изменяет размер буфера, чтобы он содержал только полученные/записанные данные. Наконец, мы вызываем функцию обратного вызова, которая была первоначально передана функции async и сохранена как task_unwrapped_.



К чему все это? А к тому, что теперь у нас есть простой способ отправки данных!

Теперь у нас есть функция `async_write_data`, которая асинхронно записывает данные из буфера в сокет и выполняет обратный вызов по завершении операции:

```
void on_send(connection_ptr&& soc, const boost::system::error_code& err);

void connect_and_send() {
    connection_ptr s = tasks_processor::create_connection
        ("127.0.0.1", 80);

    s->data = "data_to_send";
    async_write_data(
        std::move(s),
        &on_send
    );
}
```

Функция `async_read_data` очень похожа на `async_write_data`. Она изменяет размер буфера, создает функцию `task_wrapped_with_connection` и помещает ее в `is_service` после завершения асинхронной операции.

Обратите внимание на функцию `async_read_data_at_least`. В ее теле содержится немного другой вызов `boost::asio::async_read`:

```
boost::asio::async_read(
    s,
    boost::asio::buffer(p, at_most),
    boost::asio::transfer_at_least(at_least_bytes),
    task_wrapped_with_connection<Functor>(std::move(c), f)
);
```

Здесь есть `boost::asio::transfer_at_least(at_least_bytes)`. В `Boost.Asio` имеется множество функторов для настройки операций чтения и записи. Этот функтор говорит, что *перед выполнением функции обратного вызова прочитайте как минимум* `at_least_bytes` *байт. Если байтов больше, то это нормально, пока они помещаются в буфер.*

Наконец, давайте посмотрим на одну из функций обратного вызова:

```
void process_server_response(
        connection_ptr&& soc,
        const boost::system::error_code& err);
```

В этом примере функции обратного вызова должны принимать `connection_ptr` и переменную `boost::system::error_code`. Переменная `boost::system::error_code` содержит информацию об ошибках. У нее имеется явное преобразование в оператор `bool`, поэтому если вы хотите выполнить проверку на наличие ошибок – просто напишите: `if (err) { ... }`. Если удаленное устройство завершает передачу и закрывает сокет, `err` может содержать код ошибки `boost::asio::error::eof`, что не всегда плохо. В нашем примере мы рассматриваем это как поведение без ошибок:

```
if (err && err != boost::asio::error::eof) {
    std::cerr << "Client error on receive: "
              << err.message() << '\n';
    assert(false);
}
```

Поскольку мы связали сокет и буфер, полученные данные вам доступны из soc->data:

```
if (soc->data.size() != 2) {
    std::cerr << "Wrong bytes count\n";
    assert(false);
}

if (soc->data != "OK") {
    std::cerr << "Wrong response: " << soc->data << '\n';
    assert(false);
}
```

Вызов soc->shutdown() является необязательным в нашем примере, потому что, когда soc выходит за пределы области видимости, для него вызывается деструктор. Деструктор unique_ptr<connection_with_data> вызывает ~connection_with_data, в теле которой есть функция shutdown().

Дополнительно...

Наша функция task_wrapped_with_connection::operator() недостаточно хороша! Созданная пользователем функция обратного вызова call_unwrapped_ может выбрасывать исключения и может получить прерывание из Boost.Thread, которое не принадлежит этой конкретной задаче. Чтобы это исправить, можно обернуть функцию обратного вызова в класс из первого рецепта:

```
void operator()(
    const boost::system::error_code& error,
    std::size_t bytes_count)
{
    const auto lambda = [this, &error, bytes_count]() {
        this->c_->data.resize(bytes_count);
        this->task_unwrapped_(std::move(this->c_), error);
    };

    const auto task = detail::make_task_wrapped(lambda);
    task();
}
```

В task_wrapped_with_connection::operator() мы создаем лямбда-функцию с именем lambda. При выполнении она изменяет размер data внутри класса connection_with_data до bytes_count и запускает первоначально переданную функцию обратного вызова. Наконец, мы оборачиваем функцию lambda в наш безопасный класс для выполнения задач из первого рецепта, а затем выполняем ее.

Можно увидеть множество примеров Boost.Asio в интернете. Многие из них для хранения данных используют shared_ptr вместо unique_ptr. Подход с применением shared_ptr проще реализовать; однако у него есть два больших недостатка:

- эффективность: внутри shared_ptr имеется атомарный счетчик, и если изменять его из разных потоков, это может значительно снизить производительность. В одном из последующих рецептов вы увидите, как обрабатывать задачи в нескольких потоках, и именно с многопоточной обработкой различия могут быть заметны в случаях высокой нагрузки;
- явность: в случае с unique_ptr всегда видно, что владение соединением было передано куда-то (вы видите std::move в коде). В случае с shared_ptr нельзя понять из интерфейса, получает ли функция объект во владение или просто использует ссылку на объект.

Однако, возможно, вы будете вынуждены использовать shared_ptr, если в соответствии с логикой вашего приложения владение должно распределяться между несколькими задачами одновременно.

Библиотека Boost.Asio не является частью C++, но вскоре будет поставляться как часть Networking TS и будет включена в один из последующих стандартов C++.

См. также

- Посетите страницу http://boost.org/libs/asio, где приводятся дополнительные примеры, учебные пособия, полная справочная документация и пример использования протоколов UDP или ICMP;
- вы также можете прочитать книгу *Boost.Asio C++ Network Programming*, в которой Boost.Asio описывается более подробно.

Прием входящих соединений

Работа с сетью на стороне сервера часто выглядит как последовательность, в которой мы сначала получаем новое соединение, читаем данные, затем обрабатываем их и отправляем результат. Представьте, что мы создаем какой-нибудь сервер авторизации, который должен обрабатывать огромное количество запросов в секунду. В этом случае нам нужно асинхронно принимать, отправлять и обрабатывать задачи в нескольких потоках.

В этом рецепте мы увидим, как расширить наш класс tasks_processor, чтобы принимать и обрабатывать входящие соединения, а в следующем рецепте узнаем, как сделать его многопоточным.

Подготовка

Этот рецепт требует хорошего знания основ класса boost::asio::io_service, который описан в первых рецептах данной главы. Знания о передаче данных по сети будут полезны. Также необходимо знание boost::function и информации как минимум из двух предыдущих рецептов. Линкуйте этот рецепт с библиотеками boost_system и boost_thread. Определите макрос BOOST_ASIO_DISABLE_HANDLER_TYPE_REQUIREMENTS для обхода устаревших проверок библиотеки Boost.Asio.

Как это делается...

Как и в предыдущих рецептах, мы добавляем в наш класс tasks_processor новые методы.

1. Начнем с добавления псевдонимов типов:

```
class tasks_processor {
    typedef boost::asio::ip::tcp::acceptor acceptor_t;

    typedef boost::function<
        void(connection_ptr, const boost::system::error_code&)
    > on_accept_func_t;
```

2. Давайте добавим класс, который связывает сокет для получения новых входящих соединений, сокет нового соединения и предоставляемую пользователем функцию обратного вызова для обработки новых соединений:

```
private:
    struct tcp_listener {
        acceptor_t acceptor_;
        const on_accept_func_t func_;
        connection_ptr new_c_;

        template <class Functor>
        tcp_listener(
                boost::asio::io_service& io_service,
                unsigned short port,
                const Functor& task_unwrapped)
            : acceptor_(io_service, boost::asio::ip::tcp::endpoint(
                boost::asio::ip::tcp::v4(), port
            ))
            , func_(task_unwrapped)
        {}
    };

    typedef std::unique_ptr<tcp_listener> listener_ptr;
```

3. Нам нужно добавить функцию, которая начинает слушать указанный порт:

```
public:
    template <class Functor>
    static void add_listener(unsigned short port_num, const Functor& f) {
        std::unique_ptr<tcp_listener> listener(
            new tcp_listener(get_ios(), port_num, f)
        );

        start_accepting_connection(std::move(listener));
    }
```

4. Функция, которая начинает принимать входящие соединения:

```
private:
   static void start_accepting_connection(listener_ptr&& listener) {
     if (!listener->acceptor_.is_open()) {
         return;
     }
```

```
        listener->new_c_.reset(new connection_with_data(
            listener->acceptor_.get_io_service()
        ));

        boost::asio::ip::tcp::socket& s = listener->new_c_->socket;
        acceptor_t& a = listener->acceptor_;
        a.async_accept(
            s,
            tasks_processor::handle_accept(std::move(listener))
        );
    }
```

5. Нам также нужен функтор, который обрабатывает новое соединение:

```
    private:
        struct handle_accept {
            listener_ptr listener;

            explicit handle_accept(listener_ptr&& l)
                : listener(std::move(l))
            {}

            void operator()(const boost::system::error_code& error) {
                task_wrapped_with_connection<on_accepet_func_t> task(
                    std::move(listener->new_c_), listener->func_
                );

                start_accepting_connection(std::move(listener));
                task(error, 0);
            }
        };
```

Готово! Теперь мы можем принять соединение:

```
class authorizer {
public:
    static void on_connection_accept(
        connection_ptr&& connection,
        const boost::system::error_code& error)
    {
        assert(!error);
        // ...
    }
};

int main() {
    tasks_processor::add_listener(80, &authorizer::on_connection_accept);
    tasks_processor::start();
}
```

Как это работает...

Функция add_listener создает новый экземпляр класса tcp_listener, в котором хранятся все данные, необходимые для принятия соединений. Как и в случае

любой асинхронной операции, нам нужно следить за временем жизни ресурсов в процессе выполнения операций. Уникальный указатель на `tcp_listener` как раз выполняет эту работу.

Когда мы создаем `boost::asio::ip::tcp::acceptor` с указанием конечной точки `boost::asio::ip::tcp::endpoint` (*см. шаг 3*), он открывает сокет по указанному адресу и готовится к принятию соединений.

В *шаге 4* мы создаем новый сокет и вызываем `async_accept` для этого нового сокета. Когда приходит новое соединение, `listener->acceptor_` связывает это соединение с переданным сокетом и помещает функцию обратного вызова `tasks_processor::handle_accept` в `boost::asio::io_service`. Как мы поняли из предыдущего рецепта, все вызовы `async_*` возвращаются незамедлительно, и `async_accept` не является исключением.

Давайте внимательнее посмотрим на наш `handle_accept::operator()`. В нем мы создаем функтор `task_wrapped_with_connection` из предыдущего рецепта и перемещаем в него новое соединение. Теперь у нашего `listener_ptr` нет сокета в `new_c_`, так как он принадлежит функтору. Мы вызываем функцию `start_accepting_connection(std::move(listener))`, и она создает новый сокет в `listener->new_c_` и запускает асинхронное принятие новых соединений. Асинхронная операция принятия не блокируется, поэтому программа продолжает выполнение, возвращается из функции `start_accepting_connection(std::move(listener))` и выполняет функтор с соединением `task(error, 0)`.

Вы сделали все, как показано в примере, но производительность сервера недостаточно высока? Это связано с тем, что пример упрощен и многие оптимизации остались позади. Наиболее важной из них является создание отдельного небольшого буфера в `connection_with_data` и использование его для всех внутренних выделений памяти, связанных с обратными вызовами `Boost.Asio`. См. пример пользовательского распределения памяти (англ. *Custom memory allocation example*) в официальной документации библиотеки `Boost.Asio` для получения дополнительной информации об этой оптимизации.

Когда вызывается деструктор для `boost::asio::io_service`, также вызываются деструкторы для всех функций обратного вызова. Это приводит к вызову деструктора для `tcp_connection_ptr` и освобождению ресурсов.

Дополнительно...

Мы не использовали все возможности класса `boost::asio::ip::tcp::acceptor`. Он может связываться с конкретным адресом IPv6 или IPv4, если мы предоставляем конкретную точку `boost::asio::ip::tcp::endpoint`. Вы также можете получить нативный сокет с помощью метода `native_handle()` и использовать некоторые специфичные для ОС вызовы для настройки поведения. Можно настроить параметры для `acceptor_`, вызвав `set_option`. Например, вот как можно заставить `acceptor_` использовать адрес повторно:

```
boost::asio::socket_base::reuse_address option(true);
acceptor_.set_option(option);
```

Повторное использование адреса дает возможность быстро перезапустить сервер после того, как он аварийно завершил работу. После аварийного завершения работы сервера сокет может быть открыт в течение некоторого времени, и вы не сможете запустить сервер по тому же адресу без опции `reuse_address`.

В стандарте C++ нет классов из `Boost.Asio`, но Networking TS с большей частью функциональности может скоро появиться в ваших стандартных библиотеках.

См. также

- Неплохо будет начать читать эту главу с самого начала, чтобы получить гораздо больше информации о библиотеке `Boost.Asio`;
- *см.* официальную документацию по `Boost.Asio` для получения дополнительных примеров и учебных пособий на странице http://boost.org/libs/asio.

Параллельное выполнение различных задач

Теперь настало время заставить ваш класс `task_processor` обрабатывать задачи в нескольких потоках. Насколько это сложно?

Приступаем...

Вам нужно будет прочитать первый рецепт из этой главы. Также требуются знания о многопоточности, особенно чтение рецепта «*Управление группой потоков*».

Линкуйте этот рецепт с библиотеками `boost_system` и `boost_thread`. Определите макрос `BOOST_ASIO_DISABLE_HANDLER_TYPE_REQUIREMENTS`, чтобы обойти устаревшие проверки библиотеки `Boost.Asio`.

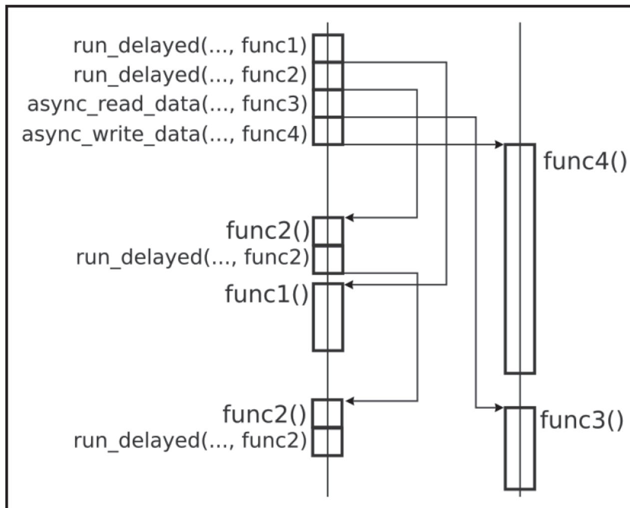
Как это делается...

Все, что нам нужно сделать, – это добавить метод `start_multiple` в наш класс `tasks_processor`:

```
#include <boost/thread/thread.hpp>

class tasks_processor {
public:
    // Со значением по умолчанию мы попытаемся угадать оптимальное количество потоков
    static void start_multiple(std::size_t threads_count = 0) {
        if (!threads_count) {
            threads_count = (std::max)(static_cast<int>(
                boost::thread::hardware_concurrency()), 1
            );
        }

        // Первый поток является текущим
        -- threads_count;

        boost::asio::io_service& ios = get_ios();
```

```
        boost::thread_group tg;
        for (std::size_t i = 0; i < threads_count; ++i) {
            tg.create_thread([&ios]() { ios.run(); });
        }

        ios.run();
        tg.join_all();
    }
};
```

А теперь мы можем проделать гораздо больше работы, как показано на приведенной ниже диаграмме.



Как это работает...

Метод `boost::asio::io_service::run` является потокобезопасным. Все, что нам нужно сделать, – просто запустить его из разных потоков.

Если вы выполняете задачи, которые изменяют общий ресурс, вам нужно добавить мьютексы вокруг этих ресурсов или организовать свое приложение таким образом, чтобы общий ресурс не использовался одновременно разными задачами. Безопасно передавать ресурс из задачи в задачу без одновременного доступа к ресурсу, потому что `boost::asio::io_service` заботится о дополнительной синхронизации задач и делает так, чтобы результаты модификации одного задания были видны другому заданию.

Посмотрите на вызов функции `boost::thread::hardware_concurrency()`. Он возвращает количество потоков, которые могут работать одновременно на текущем оборудовании. Но это всего лишь подсказка, и иногда может возвращаться значение `0`, поэтому мы вызываем для него функцию `std::max`. Она гарантирует, что `threads_count` хранит хотя бы значение 1.

Мы заключили `std::max` в круглые скобки, потому что некоторые популярные компиляторы определяют макросы `min()` и `max()`. Скобки помогают вызвать именно функции, а не макросы.

Дополнительно...

Функция `boost::thread::hardware_concurrency()` является частью стандарта C++11; ее можно найти в заголовке `<thread>` пространства имен `std::`.

Ни один из классов `boost::asio` пока не является частью стандарта C++, но скоро они будут доступны в качестве Networking TS.

См. также

- *См.* документацию по `Boost.Asio` для получения дополнительных примеров и информации о различных классах http://boost.org/libs/asio;
- рецепты из главы 5 «*Многопоточность*» (в особенности последний рецепт «*Управление группой потоков*») дадут вам информацию об использовании `Boost.Thread`;
- *см.* документацию по `Boost.Thread` для получения информации о `boost::thread_group` и `boost::threads` на странице http://boost.org/libs/thread.

Конвейерная обработка задач

Иногда возникает необходимость обрабатывать задачи за заданный промежуток времени. По сравнению с предыдущими рецептами, где мы пытались обрабатывать задачи в порядке их появления в очереди, это большая разница.

Рассмотрим программу, которая соединяет две подсистемы, одна из которых создает пакеты данных, а другая записывает измененные данные на диск (что-то подобное можно увидеть в видеокамерах, устройствах для записи звука и др.). Нам нужно обрабатывать пакеты данных один за другим в указанном порядке, плавно с малыми задержками и в нескольких потоках.

Примитивный подход здесь не работает:

```
#include <boost/thread/thread.hpp>

subsystem1 subs1;
subsystem2 subs2;

void process_data() {
    while (!subs1.is_stopped()) {
        data_packet data = subs1.get_data();
        decoded_data d_decoded = decode_data(data);
        compressed_data c_data = compress_data(d_decoded);
        subs2.send_data(c_data);
     }
}

void run_in_multiple_threads() {
    boost::thread t(&process_data);
```

```
    process_data();

    t.join();
}
```

В многопоточной среде мы можем начать обрабатывать *пакет № 1* в первом потоке, а затем *пакет № 2* во втором потоке выполнения. Из-за разного времени обработки, переключений контекстов ОС и особенностей планирования задач в ОС *пакет № 2* может быть обработан до *пакета № 1*.

Нет гарантии для пакетов и порядка обработки. Давайте это исправим!

Подготовка

Для понимания этого примера необходим рецепт «*Создание класса work_queue*» главы 5 «*Многопоточность*». Код должен линковаться с библиотеками `boost_thread` и `boost_system`. Требуются базовые знания C++11, в особенности знание лямбда-функций.

Как это делается...

Этот рецепт основан на коде класса `work_queue` из рецепта «*Создание класса work_queue*» главы 5 «*Многопоточность*». Мы выполним некоторые модификации и будем использовать несколько экземпляров этого класса.

1. Начнем с создания отдельных очередей для декодирования данных, их сжатия и отправки:

   ```
   work_queue decoding_queue, compressing_queue, sending_queue;
   ```

2. Теперь пришло время провести рефакторинг `process_data` и разделить его на несколько функций:

   ```
   void start_data_accepting();
   void do_decode(const data_packet& packet);
   void do_compress(const decoded_data& packet);

   void start_data_accepting() {
       while (!subs1.is_stopped()) {
           data_packet packet = subs1.get_data();

           decoding_queue.push_task(
               [packet]() {
                   do_decode(packet);
               }
           );
       }
   }

   void do_decode(const data_packet& packet) {
       decoded_data d_decoded = decode_data(packet);

       compressing_queue.push_task(
           [d_decoded]() {
               do_compress(d_decoded);
           }
   ```

```
        );
    }

    void do_compress(const decoded_data& packet) {
        compressed_data c_data = compress_data(packet);
        sending_queue.push_task(
            [c_data]() {
                subs2.send_data(c_data);
            }
        );
    }
```

3. Наш класс `work_queue` из главы 5 «*Многопоточность*» получает функции для остановки и выполнения задач:

```
#include <deque>
#include <boost/function.hpp>
#include <boost/thread/mutex.hpp>
#include <boost/thread/locks.hpp>
#include <boost/thread/condition_variable.hpp>

class work_queue {
public:
    typedef boost::function<void()> task_type;

private:
    std::deque<task_type>      tasks_;
    boost::mutex               mutex_;
    boost::condition_variable  cond_;
    bool                       is_stopped_;

public:
    work_queue()
        : is_stopped_(false)
    {}

    void run();
    void stop();
    // То же, что и в главе 5, но с поддержкой rvalue-ссылок.
    void push_task(task_type&& task);
};
```

4. Реализация функций `stop()` и `run()` из класса `work_queue` должна выглядеть следующим образом:

```
void work_queue::stop() {
    boost::lock_guard<boost::mutex> lock(mutex_);
    is_stopped_ = true;
    cond_.notify_all();
}
```

```
void work_queue::run() {
    while (1) {
        boost::unique_lock<boost::mutex> lock(mutex_);
        while (tasks_.empty()) {
            if (is_stopped_) {
                return;
            }
            cond_.wait(lock);
        }

        task_type t = std::move(tasks_.front());
        tasks_.pop_front();
        lock.unlock();
        t();
    }
}
```

5. На этом все! Теперь нам нужно только запустить конвейер:

```
#include <boost/thread/thread.hpp>
int main() {
    boost::thread t_data_decoding(
        []() { decoding_queue.run(); }
    );
    boost::thread t_data_compressing(
        []() { compressing_queue.run(); }
    );
    boost::thread t_data_sending(
        []() { sending_queue.run(); }
    );

    start_data_accepting();
```

6. Остановить его можно так:

```
decoding_queue.stop();
t_data_decoding.join();

compressing_queue.stop();
t_data_compressing.join();

sending_queue.stop();
t_data_sending.join();
```

Как это работает...

Хитрость заключается в том, чтобы разделить обработку одного пакета данных на несколько одинаково небольших подзадач и обрабатывать их одну за другой в разных `work_queues`. В этом примере мы можем разделить процесс обработки данных на декодирование данных, сжатие и отправку.

В идеале обработка шести пакетов должна выглядеть так:

Время	Получение	Декодирование	Сжатие	Отправка
Такт 1	пакет № 1			
Такт 2	пакет № 2	пакет № 1		
Такт 3	пакет № 3	пакет № 2	пакет № 1	
Такт 4	пакет № 4	пакет № 3	пакет № 2	пакет № 1
Такт 5	пакет № 5	пакет № 4	пакет № 3	пакет № 2
Такт 6	пакет № 6	пакет № 5	пакет № 4	пакет № 3
Такт 7		пакет № 6	пакет № 5	пакет № 4
Такт 8			пакет № 6	пакет № 5

Однако наш мир не идеален, поэтому некоторые задачи могут заканчиваться быстрее. Например, получение данных может работать быстрее, чем декодирование, и в этом случае очередь декодирования будет содержать набор задач, которые необходимо выполнить. Чтобы избежать переполнения очереди, постарайтесь делать так, чтобы каждая последующая обработка выполнялась немного быстрее предыдущей.

В нашем примере мы не использовали `boost::asio::io_service`, потому что он не гарантирует, что опубликованные задачи выполняются в порядке их отправки.

Дополнительно...

Все инструменты, использованные для создания конвейера в этом примере, доступны в C++11, поэтому ничто не помешает вам создавать те же вещи без Boost на компиляторе, совместимом с C++11. Тем не менее Boost делает ваш код более переносимым и удобным для использования на компиляторах для стандарта, предшествующего C++11.

См. также

- Этот метод хорошо известен и используется разработчиками процессоров. Здесь вы можете найти краткое описание всех характеристик конвейера: http://en.wikipedia.org/wiki/Instruction_pipeline;
- рецепт «*Создание класса work_queue*» главы 5 «*Многопоточность*» даст вам больше информации о методах, используемых в этом рецепте.

Создание неблокирующего барьера

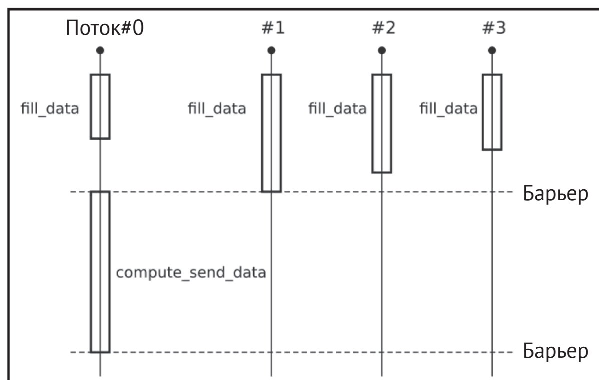
В многопоточном программировании существует абстракция под названием **барьер**. Он останавливает достигающие его потоки выполнения, пока запрошенное число потоков не будет заблокировано. После этого все потоки освобождаются и продолжают выполнение. Рассмотрим приведенный ниже пример, где его можно использовать.

Нам нужно обработать разные части данных в разных потоках и затем отправить их:

```
#include <boost/array.hpp>
#include <boost/thread/barrier.hpp>
#include <boost/thread/thread.hpp>

typedef boost::array<std::size_t, 10000> vector_type;
```

```
typedef boost::array<vector_type, 4> data_t;

void fill_data(vector_type& data);
void compute_send_data(data_t& data);

void runner(std::size_t thread_index, boost::barrier& barrier, data_t& data) {
    for (std::size_t i = 0; i < 1000; ++ i) {
        fill_data(data.at(thread_index));
        barrier.wait();

        if (!thread_index) {
            compute_send_data(data);
        }
        barrier.wait();
    }
}

int main() {
    // Инициализация барьера.
    boost::barrier barrier(data_t::static_size);

    // Инициализация данных.
    data_t data;

    // Запуск в 4 потоках.
    boost::thread_group tg;
    for (std::size_t i = 0; i < data_t::static_size; ++i) {
        tg.create_thread([i, &barrier, &data] () {
            runner(i, barrier, data);
        });
    }

    tg.join_all();
}
```

Вызов метода `data_barrier.wait()` блокирует поток выполнения, пока его не позовут еще три потока. После этого все потоки освобождаются и продолжают свое выполнение. Поток с индексом `0` вычисляет данные для отправки, используя `compute_send_data(data)`, в то время как другие потоки снова ждут у барьера, как показано на этой диаграмме:



Выглядит не очень, не так ли?

Подготовка

Этот рецепт требует знания первого рецепта этой главы. Также требуется знание библиотеки `Boost.Thread`. Код из этого рецепта требует линковки с библиотеками `boost_thread` и `boost_system`.

Как это делается...

Не нужно ничего блокировать! Давайте внимательнее посмотрим на этот пример. Все, что нам нужно сделать, – это опубликовать четыре задачи `fill_data` и сделать, чтобы последний отработавший `fill_data` выполнил задачу `compute_send_data(data)`.

1. Нам понадобится класс `tasks_processor` из первого рецепта; никаких изменений в нем делать не нужно.
2. Вместо барьера мы будем использовать атомарную переменную:

```
#include <boost/atomic.hpp>
typedef boost::atomic<unsigned int> atomic_count_t;
```

3. Наша новая функция `clever_runner` будет выглядеть так:

```
void clever_runner(
        std::size_t thread_index,
        std::size_t iteration,
        atomic_count_t& counter,
        data_t& data)
{
    fill_data(data.at(thread_index));
    if (++counter != data_t::static_size) {
        return;
    }

    compute_send_data(data);

    if (++iteration == 1000) {
        // Мы совершили 1000 итераций и выходим.
        tasks_processor::stop();
        return;
    }

   counter = 0;
   for (std::size_t i = 0; i < data_t::static_size; ++ i) {
       tasks_processor::push_task([i, iteration, &counter, &data]() {
           clever_runner(
               i,
               iteration,
               counter,
               data
           );
       });
   }
}
```

4. Функция `main` требует незначительного изменения:

```
// Инициализация счетчика.
atomic_count_t counter(0);

// Инициализация данных.
data_t data;

// Запуск четырех задач.
for (std::size_t i = 0; i < data_t::static_size; ++i) {
    tasks_processor::push_task([i, &counter, &data]() {
        clever_runner(
            i,
            0, // первая итерация
            counter,
            data
        );
    });
}

tasks_processor::start();
```

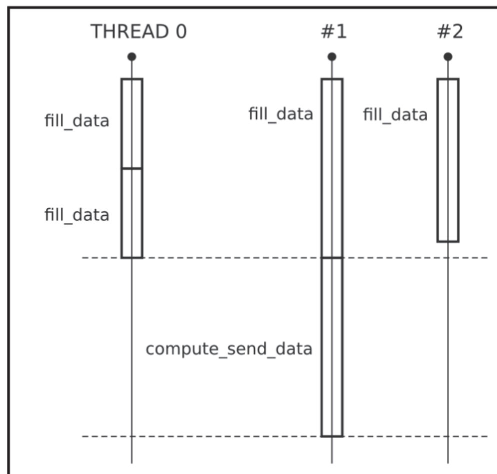
Как это работает...

Мы ничего не блокируем. Вместо этого мы считаем задачи, которые закончили заполнять данные. Это делается с помощью атомарной переменной `counter`. Мы увеличиваем ее значение, получаем копию нового значения и сравниваем его. Последняя оставшаяся задача будет иметь переменную копию значения `counter`, равную `data_t::static_size`. Только эта задача должна вычислять и отправлять данные.

После этого мы проверяем условие выхода (выполнилась ли 1000 итераций) и публикуем новые данные, помещая задачи в очередь.

Дополнительно...

Более ли это подходящее решение? Ну, во-первых, оно лучше масштабируется.



Такой подход также может быть более эффективным в ситуациях, когда программа выполняет много другой работы. Поскольку в барьерах потоки не ожидают, свободные потоки могут выполнять другие задачи, пока один из потоков вычисляет и отправляет данные.

Этот рецепт можно реализовать в C++11 без библиотек Boost. Вам нужно будет только заменить `io_service` внутри класса `tasks_processor` на `work_queue` из главы 5 «*Многопоточность*». Но, как всегда, Boost обеспечивает лучшую переносимость, и этот пример можно запускать на компиляторах, использующих библиотеки Boost, для стандарта, предшествующего C++11. Вам только нужно будет заменить лямбда-функции на `boost::bind` и `boost::ref`.

См. также

- Официальная документация по `Boost.Asio` может предоставить больше информации об использовании `io_service` по адресу http://boost.org/libs/asio;
- *см.* все рецепты, связанные с `Boost.Function`, из главы 2 «*Управление ресурсами*» и официальную документацию по адресу http://boost.org/libs/function, чтобы узнать, как работают задачи;
- *см.* рецепты из главы 1 «*Приступаем к написанию приложения*», связанные с `Boost.Bind`, для получения дополнительной информации о том, что делает функция `boost::bind`, или обратитесь к официальной документации по адресу http://boost.org/libs/bind.

Хранение исключения и создание задачи из него

Обработка исключений не всегда проста и может занимать много времени. Рассмотрим ситуацию, когда исключение должно быть сериализовано и отправлено по сети. На это может уйти несколько миллисекунд и несколько тысяч строк кода. То место, где исключение перехватывается, может быть не лучшим временем и местом для его обработки.

Можно ли сохранить исключения и отложить их обработку?

Подготовка

Этот рецепт требует знакомства с классом `boost::asio::io_service`, который был описан в первом рецепте данной главы.

Этот рецепт требует линковки с библиотеками `boost_system` и `boost_thread`.

Как это делается...

Все, что нам нужно, – это иметь возможность сохранять исключения и передавать их между потоками, как обычную переменную.

1. Давайте начнем с функции, которая хранит и обрабатывает исключения:

```
#include <boost/exception_ptr.hpp>

struct process_exception {
    boost::exception_ptr exc_;

    explicit process_exception(const boost::exception_ptr& exc)
        : exc_(exc)
    {}
```

```
    void operator()() const;
};
```

2. `operator()` этого функтора просто выводит исключение в консоль:

```
#include <boost/lexical_cast.hpp>
void func_test2(); // Forward declaration.

void process_exception::operator()() const {
    try {
        boost::rethrow_exception(exc_);
    } catch (const boost::bad_lexical_cast& /*e*/) {
        std::cout << "Lexical cast exception detected\n" << std::endl;

        // Помещаем в очередь другую задачу для выполнения.
        tasks_processor::push_task(&func_test2);
    } catch (...) {
       std::cout << "Can not handle such exceptions:\n"
           << boost::current_exception_diagnostic_information()
           << std::endl;

         // Остановка.
         tasks_processor::stop();
    }
}
```

3. Давайте напишем несколько функций, чтобы продемонстрировать, как работают исключения:

```
#include <stdexcept>

void func_test1() {
    try {
        boost::lexical_cast<int>("oops!");
    } catch (...) {
        tasks_processor::push_task(
            process_exception(boost::current_exception())
        );
    }
}

void func_test2() {
    try {
        // ...
        BOOST_THROW_EXCEPTION(std::logic_error("Some fatal logic error"));
        // ...
    } catch (...) {
        tasks_processor::push_task(
            process_exception(boost::current_exception())
        );
    }
}
```

Теперь если мы запустим пример:

```
tasks_processor::get().push_task(&func_test1);
tasks_processor::get().start();
```

то получим следующий вывод:

```
Lexical cast exception detected
Can not handle such exceptions:
main.cpp(48): Throw in function void func_test2()
Dynamic exception type: boost::exception_detail::clone_impl<boost::exception_
detail::error_info_std::exception::what: Some fatal logic error
```

Как это работает...

Библиотека Boost.Exception предоставляет возможность хранить и повторно генерировать исключения. Метод boost::current_exception() должен вызываться только изнутри блока catch(), и он возвращает объект типа boost::exception_ptr.

В примере в функции func_test1() выбрасывается исключение boost::bad_lexical_cast. Оно возвращается методом boost::current_exception(); из этого исключения создается задача process_exception.

Единственный способ восстановить тип исключения из объекта boost::exception_ptr – повторно выбросить его с помощью функции boost::rethrow_exception(exc). Это и делает функция process_exception.

Выбрасывать и перехватывать исключения – непростая операция. Выбрасывание исключений может динамически выделять память, блокировать мьютекс и выполнять многие другие действия. Не выбрасывайте исключения на горячем пути вашего приложения без очень веских на то причин!

В func_test2 мы выбрасываем исключение std::logic_error, используя макрос BOOST_THROW_EXCEPTION. Этот макрос делает много полезной работы; он проверяет, что наше исключение наследуется от std::exception, добавляет к нашему исключению информацию об имени исходного файла, имени функции и номере строки кода, откуда было выброшено исключение. Когда наше исключение std::logic_error повторно генерируется внутри process_exception::operator(), оно перехватывается catch(...).

boost::current_exception_diagnostic_information() выводит как можно больше информации о сгенерированном исключении.

Дополнительно...

Обычно exception_ptr используется для передачи исключений между потоками. Например:

```
void run_throw(boost::exception_ptr& ptr) {
    try {
        // Много всякого кода.
    } catch (...) {
        ptr = boost::current_exception();
    }
}
```

```
int main () {
    boost::exception_ptr ptr;

    // Параллельное выполнение какой-либо работы.
    boost::thread t(
        &run_throw,
        boost::ref(ptr)
    );

    // Здесь идет некий код..
    // ...

    t.join();
    // Проверка на исключение.
    if (ptr) {
         // В потоке произошло исключение.
         boost::rethrow_exception(ptr);
    }
}
```

Класс `boost::exception_ptr` может многократно выделять память в куче, использует атомарные операции и реализует некоторые операции, повторно выбрасывая и перехватывая исключения. Старайтесь не использовать его без лишней необходимости.

Классы `boost::current_exception`, `boost::rethrow_exception` и `boost::exception_ptr` приняты стандартом C++11. Их можно найти в заголовке `<exception>` в пространстве имен `std::`. Макроса `BOOST_THROW_EXCEPTION` и функции `boost::current_exception_diagnostic_information()` нет в стандартах C++.

См. также

- Официальная документация по `Boost.Exception` на странице http://boost.org/libs/exception содержит много полезной информации о реализации и ограничениях. Вы также можете найти некоторые сведения, не описанные в этом рецепте (например, как добавить дополнительную информацию в уже выброшенное исключение);
- в первом рецепте из этой главы приводится информация о классе `tasks_processor`. Рецепт «*Преобразование строк в числа*» главы 3 «*Преобразование и приведение*» описывает библиотеку `Boost.LexicalCast`, которая использовалась в этом рецепте.

Получение и обработка системных сигналов в качестве задач

При написании какого-либо серверного приложения (особенно для ОС Linux) требуется перехват и обработка сигналов. Обычно все обработчики сигналов настраиваются при запуске сервера и не изменяются во время выполнения приложения.

Цель этого рецепта – сделать класс `tasks_processor` способным обрабатывать сигналы.

Подготовка

Нам понадобится код из первого рецепта этой главы, а также хорошее знание `Boost.Function`.

Этот рецепт требует линковки с библиотеками `boost_system` и `boost_thread`.

Как это делается...

Этот рецепт похож на рецепты из второй, третьей и четвертой глав: у нас есть функции `async`, ожидающие сигнала, обработчики асинхронного сигнала и вспомогательный код.

1. Начнем с подключения следующих заголовочных файлов:

```
#include <boost/asio/signal_set.hpp>
#include <boost/function.hpp>
```

2. Теперь добавляем функцию для обработки сигналов в класс `tasks_processor`:

```
protected:
    static boost::asio::signal_set& signals() {
        static boost::asio::signal_set signals_(get_ios());
        return signals_;
    }

    static boost::function<void(int)>& signal_handler() {
        static boost::function<void(int)> users_signal_handler_;
        return users_signal_handler_;
    }
```

3. Функция, которая будет вызываться при перехвате сигнала, выглядит следующим образом:

```
static void handle_signals(
        const boost::system::error_code& error,
        int signal_number)
{
     signals().async_wait(&tasks_processor::handle_signals);

     if (error) {
         std::cerr << "Error in signal handling: " << error << '\n';
     } else {
         boost::function<void(int)> h = signal_handler();
         h(signal_number);
     }
}
```

4. А теперь нам нужна функция для регистрации обработчика сигналов:

```
public:

    // Эта функция не является потокобезопасной!!
    // Она должна вызываться до всех вызовов `start()`.
    // Функцию можно вызвать только один раз.
    template <class Func>
    static void register_signals_handler(
            const Func& f,
```

```
                std::initializer_list<int> signals_to_wait)
        {
            // Проверяем, что это действительно первый вызов данной функции.
            assert(!signal_handler());

            signal_handler() = f;
            boost::asio::signal_set& sigs = signals();
            std::for_each(
                signals_to_wait.begin(),
                signals_to_wait.end(),
               [&sigs](int signal) { sigs.add(signal); }
            );

            sigs.async_wait(&tasks_processor::handle_signals);
    }
```

На этом все. Теперь мы готовы к обработке сигналов. Ниже приводится тестовая программа:

```
void accept_3_signals_and_stop(int signal) {
    static int signals_count = 0;
    assert(signal == SIGINT);

    ++ signals_count;
    std::cout << "Captured " << signals_count << " SIGINT\n";
    if (signals_count == 3) {
        tasks_processor::stop();
    }
}

int main () {
    tasks_processor::register_signals_handler(
        &accept_3_signals_and_stop,
        { SIGINT, SIGSEGV }
    );

    tasks_processor::start();
}
```

Запуск этого кода даст следующий вывод:

```
Captured 1 SIGINT
Captured 2 SIGINT
Captured 3 SIGINT
Press any key to continue . . .
```

Как это работает...

Здесь нет ничего сложного (по сравнению с некоторыми предыдущими рецептами из этой главы). Функция `register_signals_handler` запоминает номера сигналов, которые надо перехватывать и обрабатывать. Это делается с помощью вызова функции `boost::asio::signal_set::add` для каждого элемента `signals_to_wait`.

Затем sigs.async_wait запускает асинхронное ожидание сигнала и вызывает функцию tasks_processor::handle_signals при получении сигнала. Функция tasks_processor::handle_signals немедленно запускает асинхронное ожидание следующего сигнала, проверяет ошибки и, если их нет, выполняет функцию обратного вызова, предоставляя номер сигнала.

Дополнительно....

Можно сделать еще лучше! Мы можем обернуть предоставленную пользователем функцию обратного вызова в наш класс из первого рецепта, чтобы правильно обрабатывать исключения и делать другие полезные вещи из первого рецепта:

```
boost::function<void(int)> h = signal_handler();
detail::make_task_wrapped([h, signal_number]() {
    h(signal_number);
})(); // make and run task_wrapped
```

Когда требуется потокобезопасное динамическое добавление и удаление сигналов, мы можем изменить этот пример так, чтобы он выглядел как detail::timer_task из рецепта таймеров «*Создание таймеров и обработка событий таймера в качестве задач*». Когда несколько объектов boost::asio::signal_set регистрируются для ожидания одних и тех же сигналов, каждый обработчик из signal_set вызывается при получении одного сигнала.

C++ был способен обрабатывать сигналы в течение длительного времени, используя функцию signal из заголовка <csignal>. Networking TS, вероятно, будет лишен функциональности signal_set.

См. также

- Рецепт «*Хранение любого функционального объекта в переменной*» главы 2 «*Управление ресурсами*» предоставляет информацию о boost::function;
- *см.* официальную документацию по Boost.Asio для получения дополнительной информации и примеров по boost::asio::signal_set и другим функциям этой замечательной библиотеки по адресу http://boost.org/libs/asio.

Глава 7

Манипулирование строками

Темы, которые мы рассмотрим в этой главе:

- смена регистра символов и сравнение без учета регистра;
- сопоставление строк с использованием регулярных выражений;
- поиск и замена строк с использованием регулярных выражений;
- форматирование строк с использованием безопасных printf-подобных функций;
- замена и стирание строк;
- представление строки двумя итераторами;
- использование типа «ссылка на строку».

Вступление

Вся эта глава посвящена различным аспектам изменения, поиска и представления строк. Мы увидим, как некоторые распространенные задачи, связанные со строками, можно легко выполнять с помощью библиотек Boost. Это достаточно простая глава. Итак, начнем!

Смена регистра символов и сравнение без учета регистра

Это довольно распространенная задача. У нас есть две строки с символами ANSI:

```
#include <string>
std::string str1 = "Thanks for reading me!";
std::string str2 = "Thanks for reading ME!";
```

Нам нужно сравнить их без учета регистра. Есть много способов сделать это. Давайте посмотрим на способ, предлагаемый Boost.

Подготовка

Все, что нам нужно здесь, – базовые знания `std::string`.

Как это делается...

Ниже приведены различные способы сравнения без учета регистра:

1. Самый простой из них:

```
#include <boost/algorithm/string/predicate.hpp>

const bool solution_1 = (
     boost::iequals(str1, str2)
);
```

2. Используя предикат из Boost и метод стандартной библиотеки:

```
#include <boost/algorithm/string/compare.hpp>
#include <algorithm>

const bool solution_2 = (
    str1.size() == str2.size() && std::equal(
        str1.begin(),
        str1.end(),
        str2.begin(),
        boost::is_iequal()
    )
);
```

3. Через строчную копию обеих строк:

```
#include <boost/algorithm/string/case_conv.hpp>

void solution_3() {
    std::string str1_low = boost::to_lower_copy(str1);
    std::string str2_low = boost::to_lower_copy(str2);
    assert(str1_low == str2_low);
}
```

4. Через заглавную копию оригинальных строк:

```
#include <boost/algorithm/string/case_conv.hpp>

void solution_4() {
    std::string str1_up = boost::to_upper_copy(str1);
    std::string str2_up = boost::to_upper_copy(str2);
    assert(str1_up == str2_up);
}
```

5. Через преобразование исходных строк в нижний регистр:

```
#include <boost/algorithm/string/case_conv.hpp>

void solution_5() {
    boost::to_lower(str1);
    boost::to_lower(str2);
    assert(str1 == str2);
}
```

Как это работает...

Из представленных способов лишь второй метод не очевиден. В нем мы сравниваем длину строк и, если они имеют одинаковую длину, сравниваем строки посимвольно, используя экземпляр предиката `boost::is_iequal`. Предикат `boost::is_iequal` сравнивает два символа без учета регистра.

Библиотека `Boost.StringAlgorithm` использует букву `i` в имени метода или класса, если этот метод не чувствителен к регистру. Например, `boost::is_iequal`, `boost::iequals`, `boost::is_iless` и др.

Дополнительно...

Каждая функция и функциональный объект библиотеки `Boost.StringAlgorithm`, которая работает с регистрами символов, принимает `std::locale`. По умолчанию (и в наших примерах) методы и классы используют созданную по умолчанию `std::locale`. Если мы много работаем со строками, создать переменную `std::locale` один раз и передать ее всем методам может быть хорошей оптимизацией. Еще одной хорошей оптимизацией было бы использование локали *C* (если это позволяет логика вашего приложения) через функцию `std::locale::classic()`:

```
// На некоторых платформах std::locale::classic() работает быстрее, чем std::locale().
boost::iequals(str1, str2, std::locale::classic());
```

Никто не запрещает вам использовать обе оптимизации.

К сожалению, в C++17 пока нет строковых функций из библиотеки `Boost.StringAlgorithm`. Все эти алгоритмы быстрые и надежные, поэтому не бойтесь использовать их в своем коде.

См. также

- Официальную документацию по библиотеке `Boost.StringAlgorithm` можно найти по адресу http://boost.org/libs/algorithm/string;
- *см.* книгу Андрея Александреску и Херба Саттера «Стандарты кодирования C++», где приводится пример того, как создать нечувствительный к регистру класс строки с помощью нескольких строк кода.

Сопоставление строк с использованием регулярных выражений

Давайте сделаем что-нибудь полезное! Часто возникает задача проверки пользовательского ввода по **регулярному выражению (regex, англ. Regular Expression)**. Проблема состоит в том, что существует множество синтаксисов регулярных выражений. Одинаковые выражения по-разному работают в за-

висимости от выбранного синтаксиса, и легко ошибиться. Еще проблема заключается в том, что длинные регулярные выражения не так просто писать.

Итак, в этом рецепте мы напишем программу, которая поддерживает различные синтаксисы регулярных выражений и проверяет, соответствуют ли им входные строки.

Приступим

Этот рецепт требует базовых знаний стандартной библиотеки. Знание синтаксиса регулярных выражений может быть полезным.

Требуется линковаться с библиотекой `boost_regex`.

Как это делается...

Этот пример состоит из нескольких строк кода в функции `main()`.

1. Для его реализации нам понадобятся следующие заголовочные файлы:

```
#include <boost/regex.hpp>
#include <iostream>
```

2. В начале программы нам нужно вывести доступные синтаксисы регулярных выражений:

```
int main() {
    std::cout
        << "Available regex syntaxes:\n"
        << "\t[0] Perl\n"
        << "\t[1] Perl case insensitive\n"
        << "\t[2] POSIX extended\n"
        << "\t[3] POSIX extended case insensitive\n"
        << "\t[4] POSIX basic\n"
        << "\t[5] POSIX basic case insensitive\n\n"
        << "Choose regex syntax: ";
```

3. Теперь правильно установите флаги в соответствии с выбранным синтаксисом:

```
boost::regex::flag_type flag;
switch (std::cin.get())
{
case '0': flag = boost::regex::perl;
    break;

case '1': flag = boost::regex::perl|boost::regex::icase;
    break;
case '2': flag = boost::regex::extended;
    break;

case '3': flag = boost::regex::extended|boost::regex::icase;
    break;
case '4': flag = boost::regex::basic;
    break;
case '5': flag = boost::regex::basic|boost::regex::icase;
    break;
```

```
    default:
        std::cout << "Incorrect number of regex syntax. Exiting...\n";
        return 1;
    }

    // Отключение исключений.
    flag |= boost::regex::no_except;
```

4. Теперь мы запрашиваем шаблоны регулярных выражений в цикле:

```
    // Restoring std::cin.
    std::cin.ignore();
    std::cin.clear();
    std::string regex, str;
    do {
        std::cout << "Input regex: ";
        if (!std::getline(std::cin, regex) || regex.empty()) {
            return 0;
        }

        // Без флага `boost::regex::no_except`
        // этот конструктор может выбросить исключение
        const boost::regex e(regex, flag);
        if (e.status()) {
            std::cout << "Incorrect regex pattern!\n";
            continue;
        }
```

5. Получаем строку для сопоставления в цикле:

```
        std::cout << "String to match: ";
        while (std::getline(std::cin, str) && !str.empty()) {
```

6. Применяем к ней регулярное выражение и выводим результат:

```
            const bool matched = boost::regex_match(str, e);
            std::cout << (matched ? "MATCH\n" : "DOES NOT MATCH\n");
            std::cout << "String to match: ";
        } // Конец `while (std::getline(std::cin, str))`
```

7. Мы закончим наш пример, восстановив `std::cin` и запросив новые шаблоны регулярных выражений:

```
            // Восстановление std::cin.
            std::cin.ignore();
            std::cin.clear();
        } while (1);
    } // конец функции main()
```

Теперь, если мы выполним предыдущий пример, получим следующий вывод:

```
Available regex syntaxes:
[0] Perl
[1] Perl case insensitive
[2] POSIX extended
[3] POSIX extended case insensitive
```

```
[4] POSIX basic
[5] POSIX basic case insensitive
  Choose regex syntax: 0
   Input regex: (\d{3}[#-]){2}
   String to match: 123-123#
   MATCH
   String to match: 312-321-
   MATCH
   String to match: 21-123-
   DOES NOT MATCH
   String to match: ^Z
   Input regex: \l{3,5}
   String to match: qwe
   MATCH
   String to match: qwert
   MATCH
   String to match: qwerty
   DOES NOT MATCH
   String to match: QWE
   DOES NOT MATCH
   String to match: ^Z
   Input regex: ^Z
   Press any key to continue . . .
```

Как это работает...

Все сопоставление выполняется классом `boost::regex`. Он создает объект, который может выполнять разбор и компиляцию регулярных выражений. Дополнительные параметры конфигурации передаются в класс с помощью переменной `flag`.

Если регулярное выражение неверно, `boost::regex` выбрасывает исключение. Но если перед этим был передан флаг `boost::regex::no_except`, то об ошибке в регулярном выражении нам сообщит ненулевой результат вызова `status()` (как в нашем примере):

```
if (e.status()) {
    std::cout << "Incorrect regex pattern!\n";
    continue;
}
```

Пример:

```
Input regex: (incorrect regex(
Incorrect regex pattern!
```

Сопоставление регулярных выражений выполняется путем вызова функции `boost::regex_match`. В случае успешного сопоставления она возвращает значение `true`. В `regex_match` можно передать дополнительные флаги, но мы не стали их использовать для краткости примера.

Дополнительно...

C++11 содержит почти все классы и флаги `Boost.Regex`. Их можно найти в заголовке `<regex>` пространства имен `std::` (вместо `boost::`). Официальная доку-

ментация содержит информацию о различиях C++11 и `Boost.Regex`, а также показатели производительности, которые говорят о том, что `Boost.Regex` работает быстро. У некоторых стандартных библиотек имеются проблемы с производительностью, поэтому поступайте разумно, выбирая между версиями библиотек Boost и стандартными библиотеками.

См. также

- Рецепт «*Поиск и замена строк с помощью регулярных выражений*» даст вам больше информации об использовании библиотеки `Boost.Regex`;
- вы также можете прочитать официальную документацию для получения дополнительной информации о флагах, показателях производительности, синтаксисах регулярных выражений и соответствии C++11 по адресу http://boost.org/libs/regex.

Поиск и замена строк с использованием регулярных выражений

Моей жене очень понравился рецепт «*Сопоставление строк с помощью регулярных выражений*». Но ей захотелось большего, и она сказала мне, что я не получу еды, пока не улучшу рецепт, чтобы можно было заменять части входной строки в соответствии с регулярным выражением.

Хорошо, вот, пожалуйста!

Каждое совпадающее подвыражение (часть регулярного выражения в скобках) получает уникальный номер, начиная с 1; этот номер мы будем использовать для создания новой строки.

Вот как должна работать обновленная программа:

```
Available regex syntaxes:

        [0] Perl
        [1] Perl case insensitive
        [2] POSIX extended
        [3] POSIX extended case insensitive
        [4] POSIX basic
        [5] POSIX basic case insensitive

Choose regex syntax: 0
Input regex: (\d)(\d)
String to match: 00
MATCH: 0, 0,
Replace pattern: \1#\2
RESULT: 0#0
String to match: 42
MATCH: 4, 2,
Replace pattern: ###\1-\1-\2-\1-\1###
RESULT: ###4-4-2-4-4###
```

Подготовка

Мы будем повторно использовать код из рецепта «*Сопоставление строк с помощью регулярных выражений*». Рекомендуется прочитать его, прежде чем продолжить.

Требуется линковка с библиотекой `boost_regex`.

Как это делается...

Решение основано на предыдущем рецепте. Посмотрим, что нужно изменить.

1. Не нужно подключать никакие дополнительные заголовочные файлы. Однако нам нужна дополнительная строка для хранения шаблона замены:

```
std::string regex, str, replace_string;
```

2. Мы заменяем `boost::regex_match` на `boost::regex_find` и выводим совпадающие результаты:

```
std::cout << "String to match: ";
while (std::getline(std::cin, str) && !str.empty()) {
    boost::smatch results;
    const bool matched = regex_search(str, results, e);
    if (matched) {
        std::cout << "MATCH: ";
        std::copy(
            results.begin() + 1,
            results.end(),
            std::ostream_iterator<std::string>(std::cout, ", ")
        );
```

3. После этого нам нужно получить шаблон замены и применить его:

```
        std::cout << "\nReplace pattern: ";
        if (
            std::getline(std::cin, replace_string)
            && !replace_string.empty())
        {
            std::cout << "RESULT: " <<
                boost::regex_replace(str, e, replace_string)
            ;
        } else {
            // Восстановление std::cin.
            std::cin.ignore();
            std::cin.clear();
        }
    } else { // `if (matched) `
        std::cout << "DOES NOT MATCH";
    }
```

Вот и все! Все счастливы, а я сыт.

Как это работает...

Функция boost::regex_search не только возвращает значение true или false (в отличие от функции boost::regex_match), но также хранит совпадающие фрагменты. Мы выводим совпадающие фрагменты, используя следующую конструкцию:

```
std::copy(
    results.begin() + 1,
    results.end(),
    std::ostream_iterator<std::string>(std::cout, ", ")
);
```

Обратите внимание на то, что мы вывели результаты, пропустив первый результат (results.begin()+1), потому что results.begin() содержит всю совпавшую строку.

Функция boost::regex_replace выполняет замену и возвращает измененную строку.

Дополнительно...

Существуют различные варианты функций regex_*, некоторые из них получают двунаправленные итераторы вместо строк, а некоторые выводят результат в итератор.

boost::smatch – это typedef для boost::match_results<std::string::const_iterator>. Если вместо std::string::const_iterator вы используете другие двунаправленные итераторы, то должны передать тип двунаправленного итератора в качестве параметра шаблона для boost::match_results.

Для match_results есть функция форматирования. Мы можем подправить наш пример и вместо

```
std::cout << "RESULT: " << boost::regex_replace(str, e, replace_string);
```

использовать эту функцию форматирования:

```
std::cout << "RESULT: " << results.format(replace_string);
```

Кстати, replace_string поддерживает несколько форматов:

```
Input regex: (\d)(\d)
 String to match: 12
 MATCH: 1, 2,
 Replace pattern: $1-$2---$&---$$
 RESULT: 1-2---12---$
```

Все классы и функции из этого рецепта существуют в C++11 в пространстве имен std:: заголовочного файла <regex>.

См. также

Официальная документация по Boost.Regex предоставит вам больше примеров и информации о производительности, совместимости со стандартом C++11 и синтаксисе регулярных выражений по адресу http://boost.org/libs/regex. Рецепт «*Сопоставление строк с помощью регулярных выражений*» познакомит вас с основами Boost.Regex.

Форматирование строк с использованием безопасных printf-подобных функций

Семейство функций `printf` представляет собой угрозу безопасности. Нельзя позволять пользователям помещать свои строки, описывающие формат и тип параметров. Но что же делать, когда нам требуется создать функцию с поддержкой пользовательского форматирования? Как реализовать функцию-член `std::string to_string(const std::string& format_specifier) const;` этого класса?

```
class i_hold_some_internals
{
  int i;
  std::string s;
  char c;
  // ...
};
```

Подготовка

Базовых знаний стандартной библиотеки более чем достаточно для этого рецепта.

Как это делается...

Мы хотим разрешить пользователям указывать собственный выходной формат для строки.

1. Чтобы делать это безопасно, нам нужен следующий заголовок:

```
#include <boost/format.hpp>
```

2. Теперь добавим несколько комментариев для пользователя:

```
// В параметре `fmt` используйте:
// $1$ для вывода целого числа 'i'.
// $2$ для вывода целой строки 's'.
// $3$ для вывода целого символа 'c'.
std::string to_string(const std::string& fmt) const {
```

3. Настало время заставить все части работать:

```
   boost::format f(fmt);
   unsigned char flags = boost::io::all_error_bits;
   flags ^= boost::io::too_many_args_bit;
   f.exceptions(flags);
   return (f % i % s % c).str();
}
```

Вот и все. Посмотрите на этот код:

```
int main() {
    i_hold_some_internals class_instance;

    std::cout << class_instance.to_string(
        "Hello, dear %2%! "
```

```
        "Did you read the book for %1% %% %3%\n"
    );

    std::cout << class_instance.to_string(
        "%1% == %1% && %1%%% != %1%\n\n"
    );
}
```

Для class_instance с членом i == 100, s == "Reader" и c == '!' программа выведет следующее:

```
Hello, dear Reader! Did you read the book for 100 % !
100 == 100 && 100% != 100
```

Как это работает...

Класс boost::format принимает строку, которая определяет результирующий формат строки. Аргументы передаются в boost::format с использованием operator%. Значения %1%,%2%,%3%,%4% и т. д. в строке, определяющей формат, заменяются аргументами, передаваемыми в boost::format.

Мы также отключаем выбрасывание исключений для случаев, когда строка формата содержит меньше аргументов, чем передано в boost::format:

```
boost::format f(format_specifier);
unsigned char flags = boost::io::all_error_bits;
flags ^= boost::io::too_many_args_bit;
```

Это делается для того, чтобы допустить форматы, подобные этому:

```
// Выводит 'Reader'.
std::cout << class_instance.to_string("%2%\n\n");
```

Дополнительно...

Что происходит в случае неверного формата?

Ничего страшного, выбрасывается исключение:

```
try {
   class_instance.to_string("%1% %2% %3% %4% %5%\n");
   assert(false);
} catch (const std::exception& e) {
    // Перехватывается исключение boost::io::too_few_args.
    std::cout << e.what() << '\n';
}
```

Предыдущий фрагмент кода выведет следующие строки на консоль:

```
boost::too_few_args: format-string referred to more arguments than
were passed
```

В C++20 есть std::format, он очень производительный и не похож на boost::format. Библиотека boost.format не очень быстрая. Старайтесь реже использовать ее в местах, где важна высокая производительность.

См. также

Официальная документация содержит дополнительную информацию о производительности библиотеки `Boost.Format`. Дополнительные примеры и документация по расширенному формату, похожему на функции `printf`, доступны по адресу http://boost.org/libs/format.

Замена и стирание строк

Ситуации, когда нам нужно удалить что-либо в строке, заменить часть строки или удалить первое либо последнее вхождение какой-либо подстроки, встречаются очень часто. Стандартная библиотека C++ позволяет нам делать большую часть из перечисленного выше, но обычно для этого требуется писать слишком большое количество кода.

Мы уже видели библиотеку `Boost.StringAlgorithm` в действии в рецепте «*Смена регистра символов и сравнение без учета регистра*». Посмотрим, как можно использовать ее, чтобы упростить себе жизнь, когда нам нужно изменить какие-то строки:

```
#include <string>
const std::string str = "Hello, hello, dear Reader.";
```

Подготовка

Для этого примера необходимы базовые знания C++.

Как это делается...

Этот рецепт показывает, как работают методы стирания и замены строк из библиотеки `Boost.StringAlgorithm`.

1. Для удаления требуется заголовок `#include <boost/gorithm/string/erase.hpp>`:

```
#include <boost/algorithm/string/erase.hpp>

void erasing_examples() {
    namespace ba = boost::algorithm;
    using std::cout;

    cout << "\n erase_all_copy :" << ba::erase_all_copy(str, ",");
    cout << "\n erase_first_copy:" << ba::erase_first_copy(str, ",");
    cout << "\n erase_last_copy :" << ba::erase_last_copy(str, ",");
    cout << "\n ierase_all_copy :" << ba::ierase_all_copy(str, "hello");
    cout << "\n ierase_nth_copy :" << ba::ierase_nth_copy(str, ",", 1);
}
```

Этот код выводит следующее:

```
erase_all_copy :Hello hello dear Reader.
erase_first_copy :Hello hello, dear Reader.
erase_last_copy :Hello, hello dear Reader.
ierase_all_copy :, , dear Reader.
ierase_nth_copy :Hello, hello dear Reader.
```

2. Для замены требуется заголовок <boost/attribute/string/replace.hpp>:

```
#include <boost/algorithm/string/replace.hpp>

void replacing_examples() {
   namespace ba = boost::algorithm;
   using std::cout;

   cout << "\n replace_all_copy :"
       << ba::replace_all_copy(str, ",", "!");

   cout << "\n replace_first_copy :"
       << ba::replace_first_copy(str, ",", "!");

   cout << "\n replace_head_copy :"
       << ba::replace_head_copy(str, 6, "Whaaaaaaa!");
}
```

Этот код выводит следующее:

```
replace_all_copy :Hello! hello! dear Reader.
replace_first_copy :Hello! hello, dear Reader.
replace_head_copy :Whaaaaaaa! hello, dear Reader.
```

Как это работает...

Все примеры самодокументированы. Единственное, что неочевидно, – это функция `replace_head_copy`. Она принимает число байтов для замены в качестве второго параметра и строку замены в качестве третьего параметра. Поэтому в предыдущем примере `Hello!` заменяется на `Whaaaaaaa!`.

Дополнительно...

Есть также методы, которые изменяют строки по месту. Названия этих методов не заканчиваются на `_copy` и возвращают `void`. Все нечувствительные к регистру методы (те, что начинаются с буквы `i`) принимают `std::locale` в качестве последнего параметра и используют созданный по умолчанию `std::locale` в качестве параметра по умолчанию.

Вы часто используете методы без учета регистра и вам нужна лучшая производительность? Просто создайте переменную `std::locale` с помощью `std::locale::classic()` и передайте ее всем алгоритмам. При работе с маленькими строками большая часть времени потребляется конструкторами `std::locale`, а не самими алгоритмами:

```
#include <boost/algorithm/string/erase.hpp>

void erasing_examples_locale() {
    namespace ba = boost::algorithm;
    const std::locale loc = std::locale::classic();

    const std::string r1
        = ba::ierase_all_copy(str, "hello", loc);
```

```
    const std::string r2
        = ba::ierase_nth_copy(str, ",", 1, loc);

    // ...
}
```

В стандарте C++17 нет методов и классов библиотеки `Boost.StringAlgorithm`. Однако у него есть класс `std::string_view`, который может использовать подстроки без выделения памяти. Дополнительную информацию о классах, подобных `std::string_view`, можно найти в следующих двух рецептах этой главы.

См. также

- Множество примеров и полную справочную документацию по всем методам можно найти на странице: http://boost.org/libs/algorithm/string;
- *см.* рецепт «*Смена регистра символов и сравнение без учета регистра*» из этой главы для получения дополнительной информации о библиотеке `Boost.StringAlgorithm`.

Представление строки двумя итераторами

Бывают ситуации, когда нам нужно разбить строки на подстроки и что-то сделать с ними. В этом рецепте мы хотим разбить строку на предложения, посчитать количество символов и пробелов в каждом предложении. И конечно же, мы хотим использовать Boost и быть максимально эффективными.

Подготовка

Для этого рецепта вам понадобятся некоторые базовые знания алгоритмов стандартной библиотеки.

Как это делается...

Это очень легко сделать с помощью Boost.

1. Прежде всего подключите правильные заголовочные файлы:

```
#include <iostream>
#include <boost/algorithm/string/split.hpp>
#include <boost/algorithm/string/classification.hpp>
#include <algorithm>
```

2. Теперь давайте определим нашу тестовую строку:

```
int main() {
    const char str[] =
        "This is a long long character array."
        "Please split this character array to sentences!"
        "Do you know, that sentences are separated using period, "
        "exclamation mark and question mark? :-)"
    ;
```

3. Используем ключевое слово `typedef` для нашего итератора:

```
    typedef boost::split_iterator<const char*> split_iter_t;
```

4. Создаем итератор:

```
split_iter_t sentences = boost::make_split_iterator(str,
    boost::algorithm::token_finder(boost::is_any_of("?!."))
);
```

5. Теперь можно итерироваться по предложениям:

```
for (unsigned int i = 1; !sentences.eof(); ++sentences, ++i) {
    boost::iterator_range<const char*> range = *sentences;
    std::cout << "Sentence #" << i << " : \t" << range << '\n';
```

6. Подсчитываем количество символов:

```
    std::cout << range.size() << " characters.\n";
```

7. И пробелы:

```
    std::cout
        << "Sentence has "
        << std::count(range.begin(), range.end(), ' ')
        << " whitespaces.\n\n";
} // Конец цикла for(...)
} // Конец функции main()
```

Вот и все. Теперь если мы запустим пример, он выведет:

```
Sentence #1 : This is a long long character array
35 characters.
Sentence has 6 whitespaces.

Sentence #2 : Please split this character array to sentences
46 characters.
Sentence has 6 whitespaces.

Sentence #3 : Do you know, that sentences are separated using dot,
exclamation mark and question mark
90 characters.
Sentence has 13 whitespaces.

Sentence #4 : :-)
4 characters.
Sentence has 1 whitespaces.
```

Как это работает...

Основная идея этого рецепта заключается в том, что нам не нужно создавать `std::string` из подстрок. Нам даже не нужно разбивать на предложения всю строку сразу. Все, что нам нужно сделать, – это найти первую подстроку и вернуть ее в виде пары итераторов на начало и конец подстроки. Если нужно больше подстрок, найдите следующую подстроку и верните пару итераторов для этой подстроки.

Теперь давайте подробнее рассмотрим `boost::split_iterator`. Мы создали экземпляр этого класса, используя функцию `boost::make_split_iterator`, которая принимает `range` в качестве первого аргумента и предикат для поиска

в качестве второго аргумента. При разыменовании split_iterator возвращает первую найденную подстроку в виде boost::iterator_range<const char*>, который содержит пару указателей и имеет несколько методов для работы с ними. Когда мы выполняем операцию инкремента на split_iterator, он пытается найти следующую подстроку, и если подстрока не найдена, функция split_iterator::eof() возвращает значение true.

Созданный по умолчанию split_iterator представляет eof(). Мы могли бы переписать условие цикла из !sentences.eof() в sentences !=split_iter_t(). Вы также можете использовать split_iterator с алгоритмами, например: std::for_each(sentences, split_iter_t(), [] (auto range){ /**/ });.

Дополнительно...

Класс boost::iterator_range широко используется многими библиотеками Boost. Он может оказаться полезным даже для вашего собственного кода в ситуациях, когда необходимо вернуть пару итераторов или когда функция должна принимать или работать с парой итераторов.

Классы boost::split_iterator<> и boost::iterator_range<> принимают тип итератора в качестве параметра шаблона. Поскольку мы работали с массивом символов в предыдущем примере, мы предоставили тип const char* в качестве итераторов. Если бы мы работали с std::wstring, нам нужно было бы использовать типы boost::split_iterator<std::wstring::const_iterator> и boost::iterator_range<std::wstring::const_iterator>.

В C++20 принят класс std::ranges::subrange, во многом похожий на iterator_range. В C++20 также есть класс std::span, объявленный в заголовочном файле <span> и представляющий собой пару итераторов над непрерывной последовательностью элементов. Аналогом split_iterator в C++20 выступает std::ranges::split_view из заголовочного файла <ranges>.

Класс boost::iterator_range не имеет виртуальных функций и не использует динамическое выделение памяти, он быстрый и эффективный. Однако его оператор вывода в поток не имеет оптимизаций для массивов символов, поэтому вывод в поток может быть медленным.

Класс boost::split_iterator содержит класс boost::function, поэтому процесс его создания для больших функторов может идти медленно. Итерация добавляет лишь крошечные накладные расходы, которые вы не почувствуете даже в критичных к производительности частях вашего кода.

См. также

- Следующий рецепт расскажет вам об отличной замене boost::iterator_range<const char*>;
- официальная документация по Boost.StringAlgorithm может предоставить вам более подробную информацию о классах и целый ряд примеров на странице http://boost.org/libs/algorithm/string;
- более подробную информацию о boost::iterator_range можно найти здесь: http://boost.org/libs/range; это часть библиотеки Boost.Range, которая не описана в этой книге, но вы можете изучить ее самостоятельно.

Использование типа «ссылка на строку»

Этот рецепт является самым важным в данной главе! Давайте рассмотрим очень распространенный случай, когда мы пишем некую функцию, которая принимает строку и возвращает ее часть, «зажатую» между символами, передаваемыми в аргументах `starts` и `ends`:

```
#include <string>
#include <algorithm>

std::string between_str(const std::string& input, char starts, char ends) {
    std::string::const_iterator pos_beg
        = std::find(input.begin(), input.end(), starts);
    if (pos_beg == input.end()) {
        return std::string();
    }
    ++pos_beg;
    std::string::const_iterator pos_end
        = std::find(pos_beg, input.end(), ends);

    return std::string(pos_beg, pos_end);
}
```

Вам нравится такая реализация? На мой взгляд, она ужасна. Рассмотрим приведенный ниже вызов:

```
between_str("Getting expression (between brackets)", '(', ')');
```

В этом примере из `"Getting expression(between brackets)"` создается временная переменная `std::string`. Массив символов достаточно длинный, поэтому существует большая вероятность, что внутри конструктора `std::string` происходит динамическое выделение памяти и в эту память копируется массив символов. Затем, где-то внутри функции `between_str`, создается новая переменная `std::string`, что также может привести к еще одному динамическому выделению и копированию.

Итак, эта простая функция может и в большинстве случаев будет:

- вызывать динамическое выделение памяти (дважды);
- копировать строку (дважды);
- освобождать память (дважды).

Можно ли сделать так, чтобы это работало лучше?

Подготовка

Этот рецепт требует базовых знаний стандартной библиотеки и C++.

Как это делается...

На самом деле нам не нужен класс `std::string`. Нам нужен только какой-то легковесный класс, который не управляет ресурсами и у которого есть лишь указатель на массив символов и размер массива. Для этого в Boost есть класс `boost::string_view`.

1. Чтобы использовать класс `boost::string_view`, подключите следующий заголовочный файл:

```
#include <boost/utility/string_view.hpp>
```

2. Измените сигнатуру метода:

```
boost::string_view between(
    boost::string_view input,
    char starts,
    char ends)
```

3. Поменяйте `std::string` на `boost::string_view`: везде внутри тела функции:

```
{
   boost::string_view::const_iterator pos_beg
       = std::find(input.cbegin(), input.cend(), starts);
   if (pos_beg == input.cend()) {
       return boost::string_view();
   }
   ++ pos_beg;

   boost::string_view::const_iterator pos_end
       = std::find(pos_beg, input.cend(), ends);
    // ...
```

4. Конструктор `boost::string_view` принимает размер в качестве второго параметра, поэтому нам нужно немного изменить код:

```
   if (pos_end == input.cend()) {
       return boost::string_view(pos_beg, input.end() - pos_beg);
   }

   return boost::string_view(pos_beg, pos_end - pos_beg);
}
```

Получилось! Теперь мы можем вызвать `between("Getting expression (between brackets)", '(', ')')`, и он будет работать без динамического выделения памяти и копирования символов. И мы по-прежнему можем использовать его для `std::string`:

```
between(std::string("(expression)"), '(', ')')
```

Как это работает...

Как уже упоминалось, `boost::string_view` содержит только указатель на массив символов и размер данных. У него много конструкторов, и его можно инициализировать различными способами:

```
boost::string_view r0("^_^");
std::string O_O("O__O");
boost::string_view r1 = O_O;
std::vector<char> chars_vec(10, '#');
boost::string_view r2(&chars_vec.front(), chars_vec.size());
```

У класса `boost::string_view` есть все методы контейнера, поэтому его можно использовать с алгоритмами стандартной библиотеки и алгоритмами Boost:

```
#include <boost/algorithm/string/case_conv.hpp>
#include <boost/algorithm/string/replace.hpp>
#include <boost/lexical_cast.hpp>
#include <iterator>
#include <iostream>

void string_view_algorithms_examples() {
    boost::string_view r("O_O");
    // Находим один символ.
    std::find(r.cbegin(), r.cend(), '_');

    // Выведет 'o_o'.
    boost::to_lower_copy(std::ostream_iterator<char>(std::cout), r);
    std::cout << '\n';

    // Выведет 'O_O'.
    std::cout << r << '\n';

    // Выведет '^_^'.
    boost::replace_all_copy(
        std::ostream_iterator<char>(std::cout), r, "O", "^"
    );
    std::cout << '\n';

    r = "100";
    assert(boost::lexical_cast<int>(r) == 100);
}
```

Класс `boost::string_view` не владеет строкой, поэтому все его методы возвращают константные итераторы. Из-за этого его нельзя использовать в методах, которые изменяют данные, таких как `boost::to_lower(r)`.

При работе с `boost::string_view` мы должны дополнительно позаботиться о данных, на которые класс ссылается; они должны существовать и быть действительными в течение всего времени жизни переменной `boost::string_view`, которая на них ссылается. Если вы не можете этого гарантировать – не используйте `boost::string_view`. На практике это значит, что `boost::string_view` безопасно использовать в качестве параметров функций и локальных переменных, но нужна крайняя осторожность, если вы возвращаете `boost::string_view` из функций.

До появления Boost версии 1.61 не было класса `boost::string_view`, и вместо него использовался класс `boost::string_ref`. Эти классы очень похожи. `boost::string_view` больше соответствует дизайну C++17 и имеет лучшую поддержку constexpr. Начиная с Boost версии 1.61 `boost::string_ref` считается устаревшим.

Классы `string_view` быстрые и эффективные, потому что они никогда не выделяют память и у них нет виртуальных функций! Они предназначены для быстрой замены параметров `const std::string &` и `const char*`. Это означает, что вы можете заменить следующие три функции:

```
void foo(const std::string& s);
void foo(const char* s);
void foo(const char* s, std::size_t s_size);
```

одной:

```
void foo(boost::string_view s);
```

Дополнительно...

boost::string_view является классом C++17. Его можно найти в заголовке <string_view> в пространстве имен std::, если ваш компилятор совместим с C++17.

string_view в Boost и в стандартной библиотеке поддерживает использование в constexpr-функциях; тем не менее в настоящее время std::string_view имеет больше функций, помеченных constexpr.

Обратите внимание, что мы приняли переменную string_view по значению вместо константной ссылки. Это рекомендуемый способ передачи boost::string_view и std::string_view, потому что:

- string_view – это небольшой класс с тривиальными типами внутри. Передача его по значению обычно приводит к лучшей производительности из-за меньшего количества косвенных адресаций и позволяет компилятору выполнять больше оптимизаций;
- в других случаях, когда нет разницы в производительности, запись string_view val короче по сравнению с записью const string_view& val.

Как и std::string_view в C++17, класс boost::string_view на самом деле – это typedef:

```
typedef basic_string_view<char, std::char_traits<char> > string_view;
```

Вы также можете найти полезные следующие typedef для многобайтных символов в пространствах имен boost:: и std:::

```
typedef basic_string_view<wchar_t, std::char_traits<wchar_t> > wstring_view;

typedef basic_string_view<char16_t, std::char_traits<char16_t> > u16string_view;

typedef basic_string_view<char32_t, std::char_traits<char32_t> > u32string_view;
```

См. также

Дополнительную документацию по string_ref и string_view можно найти по адресу http://boost.org/libs/utility.

Глава 8

Метапрограммирование

Темы, которые мы рассмотрим в этой главе:

- использование типа «вектор типов»;
- манипулирование вектором типов;
- получение типа результата функции во время компиляции;
- создание метафункции высшего порядка;
- ленивое вычисление метафункции;
- преобразование всех элементов кортежа в строку;
- расщепление кортежей;
- манипулирование гетерогенными контейнерами в C++14.

Вступление

Эта глава посвящена отличным и сложным для понимания методам метапрограммирования. Данные методы не предназначены для повседневного использования, но они могут оказать реальную помощь в разработке универсальных библиотек.

В главе 4 «*Уловки времени компиляции*» уже были даны основы метапрограммирования. Рекомендуется прочитать ее для лучшего понимания. В этой главе мы пойдем дальше и посмотрим, как можно упаковать несколько типов в один тип, подобный кортежу. Мы создадим функции для управления коллекциями типов, посмотрим, как типы коллекций можно менять во время компиляции и как можно смешивать трюки времени компиляции с трюками времени выполнения. Все это – метапрограммирование.

Пристегните ремни и приготовьтесь. Поехали!..

Использование типа «вектор типов»

Бывают ситуации, когда было бы здорово работать со всеми параметрами шаблона, как если бы они находились в контейнере. Представьте, что мы пишем что-то наподобие `Boost.Variant`:

```
#include <boost/mpl/aux_/na.hpp>

// boost::mpl::na == n.a. == not available == не доступен
template <
    class T0 = boost::mpl::na,
```

```
    class T1 = boost::mpl::na,
    class T2 = boost::mpl::na,
    class T3 = boost::mpl::na,
    class T4 = boost::mpl::na,
    class T5 = boost::mpl::na,
    class T6 = boost::mpl::na,
    class T7 = boost::mpl::na,
    class T8 = boost::mpl::na,
    class T9 = boost::mpl::na
    >
struct variant;
```

Предыдущий код – это место, где начинают возникать следующие интересные задачи:

- как удалить квалификаторы `const` и `volatile` из всех типов?
- как удалить дубликаты типов?
- как получить размеры всех типов?
- как получить максимальный размер входных параметров?

Все эти задачи легко можно решить с помощью `Boost.MPL`.

Подготовка

Для этого рецепта требуются базовые знания главы 4 «*Уловки времени компиляции*». Наберитесь смелости перед чтением – в этом рецепте будет много метапрограммирования.

Как это делается…

Мы уже видели, как можно манипулировать типом во время компиляции. Почему нельзя пойти дальше и объединить несколько типов в подобие массива и выполнить операции для каждого элемента этого массива?

1. Прежде всего давайте упакуем все типы в один из контейнеров типа `Boost.MPL`:

```
#include <boost/mpl/vector.hpp>

template <
    class T0, class T1, class T2, class T3, class T4,
    class T5, class T6, class T7, class T8, class T9
>
struct variant {
    typedef boost::mpl::vector<
        T0, T1, T2, T3, T4, T5, T6, T7, T8, T9
    > types;
};
```

2. Давайте сделаем наш пример менее абстрактным и посмотрим, как он работает, если мы укажем типы:

```
#include <string>
struct declared{ unsigned char data[4096]; };
struct non_declared;
```

```
typedef variant<
    volatile int,
    const int,
    const long,
    declared,
    non_declared,
    std::string
>::types types;
```

3. Мы можем проверить все во время компиляции. Давайте проверим, что массив типов `types` не пустой:

```
#include <boost/static_assert.hpp>
#include <boost/mpl/empty.hpp>

BOOST_STATIC_ASSERT((!boost::mpl::empty<types>::value));
```

4. Мы также можем проверить, что, например, типы `non_declared` все еще находятся по индексу 4:

```
#include <boost/mpl/at.hpp>
#include <boost/type_traits/is_same.hpp>

BOOST_STATIC_ASSERT((boost::is_same<
    non_declared,
    boost::mpl::at_c<types, 4>::type
>::value));
```

5. И что последний тип все еще `std::string`:

```
#include <boost/mpl/back.hpp>

BOOST_STATIC_ASSERT((boost::is_same<
    boost::mpl::back<types>::type,
    std::string
>::value));
```

6. Мы можем провести некоторые преобразования. Начнем с удаления квалификаторов `const` и `volatile`:

```
#include <boost/mpl/transform.hpp>
#include <boost/type_traits/remove_cv.hpp>

typedef boost::mpl::transform<
    types,
    boost::remove_cv<boost::mpl::_1>
>::type noncv_types;
```

7. Вот как можно удалить дубликаты типов:

```
#include <boost/mpl/unique.hpp>

typedef boost::mpl::unique<
    noncv_types,
    boost::is_same<boost::mpl::_1, boost::mpl::_2>
>::type unique_types;
```

8. Мы можем проверить, что вектор типов содержит только пять элементов:

```
#include <boost/mpl/size.hpp>

BOOST_STATIC_ASSERT((boost::mpl::size<unique_types>::value == 5));
```

9. Вот как можно вычислить размер каждого элемента и получить вектор с размерами для каждого типа:

```
// Без этого мы получим ошибку:
// "use of undefined type 'non_declared'"
struct non_declared{};

#include <boost/mpl/sizeof.hpp>
typedef boost::mpl::transform<
    unique_types,
    boost::mpl::sizeof_<boost::mpl::_1>
>::type sizes_types;
```

10. Вот как получить максимальный размер из типа `sizes_type`:

```
#include <boost/mpl/max_element.hpp>

typedef boost::mpl::max_element<sizes_types>::type max_size_type;
```

11. Можно проверить, что максимальный размер типа равен размеру структуры `declared`, которая должна быть самой большой в нашем примере:

```
BOOST_STATIC_ASSERT(max_size_type::type::value == sizeof(declared));
```

Как это работает...

Класс `boost::mpl::vector` – это контейнер времени компиляции, содержащий типы. Если говорить более точно, это тип, который содержит типы. Мы не создаем его экземпляры; вместо этого мы просто используем его в `typedef`.

В отличие от контейнеров стандартных библиотек, у контейнеров `Boost.MPL` нет методов-членов. Вместо этого методы объявляются в отдельных заголовочных файлах. Поэтому, чтобы использовать методы, нам нужно:

1) подключить правильный заголовочный файл;
2) вызвать этот метод, обычно указав контейнер в качестве первого параметра.

Мы уже видели метафункции в главе 4 «*Уловки времени компиляции*». Мы использовали некоторые из них (такие как `boost::is_same`) из уже знакомой библиотеки `Boost.TypeTraits`.

Итак, на этапах 3, 4 и 5 мы просто вызываем метафункции для нашего типа контейнера.

Наступает самое трудное!

Заполнители (англ. **placeholders**) широко используются библиотекой `Boost.MPL` для сочетания метафункций:

```
typedef boost::mpl::transform<
    types,
    boost::remove_cv<boost::mpl::_1>
>::type noncv_types;
```

Здесь boost::mpl::_1 – это заполнитель, а все выражение означает, что для каждого типа в types нужно выполнить boost::remove_cv<>::type и вставить полученный тип в вектор результата. Получившийся вектор возвращаем с помощью ::type.

Перейдем к шагу 7. Здесь мы указываем метафункцию сравнения для boost::mpl::unique с использованием параметра шаблона boost::is_same<boost::mpl::_1, boost::mpl::_2>, где boost::mpl::_1 и boost::mpl::_2 являются заполнителями. Пример может показаться вам похожим на boost::bind(std::equal_to (), _1, _2), а все выражение на *шаге* 7 похоже на этот псевдокод:

```
std::vector<type> t; // 't'  выступает в качестве замены 'types'.
std::unique(t.begin(), t.end(), boost::bind(std::equal_to<type>(), _1, _2));
```

В *шаге* 9 есть кое-что интересное, что поможет лучшему пониманию примера. В коде sizes_types – это не вектор значений, а вектор целочисленных констант-типов, представляющих числа. sizes_types – это приведенный ниже тип:

```
struct boost::mpl::vector<
    struct boost::mpl::size_t<4>,
    struct boost::mpl::size_t<4>,
    struct boost::mpl::size_t<4096>,
    struct boost::mpl::size_t<1>,
    struct boost::mpl::size_t<32>
>
```

Последний шаг теперь должен быть ясен. Мы просто получаем максимальный элемент из size_types.

Метафункции Boost.MPL можно использовать везде, где разрешены typedef.

Дополнительно...

Использование библиотеки Boost.MPL приводит к увеличению времени компиляции, но дает вам возможность делать с типами все, что вы хотите. Ее использование не прибавляет накладных расходов времени выполнения и даже не добавит ни одной инструкции к полученному двоичному файлу. В стандарте C++ нет классов Boost.MPL, а Boost.MPL не использует возможности современного C++, такие как вариативные шаблоны. Из-за этого время компиляции Boost.MPL не настолько быстрое, насколько возможно в стандартах C++11, но делает библиотеку пригодной для использования на компиляторах C++03.

См. также

- *См.* главу 4 «*Уловки времени компиляции*», чтобы ознакомиться с основами метапрограммирования;
- рецепт *Манипулирование вектором типов* даст вам еще больше информации о метапрограммировании и библиотеке Boost.MPL;
- дополнительные примеры и полную справочную документацию можно найти на странице http://boost.org/libs/mpl.

Манипулирование вектором типов

Задача этого рецепта – изменять содержимое одного `boost::mpl::vector` в зависимости от содержимого второго `boost::mpl::vector`. Мы будем называть второй вектор вектором модификаторов, и каждый из этих модификаторов может иметь следующий тип:

```
// Сделать unsigned.
struct unsigne; // Не опечатка: `unsigned`- это ключевое слово, мы не можем
                // его использовать.

// Сделать const.
struct constant;

// Не менять тип.
struct no_change;
```

Итак, с чего начать?

Подготовка

Требуются базовые знания `Boost.MPL`. Может помочь чтение рецепта «*Использование типа “вектор типов”*» и главы 4 «*Уловки времени компиляции*».

Как это делается...

Этот рецепт похож на предыдущий, но он еще использует и условные операторы времени компиляции. Приготовьтесь, будет нелегко!

1. Начнем с заголовков:

```
// Это нам понадобится на этапе 3.
#include <boost/mpl/size.hpp>
#include <boost/type_traits/is_same.hpp>
#include <boost/static_assert.hpp>

// Это нам понадобится на этапе 4.
#include <boost/mpl/if.hpp>
#include <boost/type_traits/make_unsigned.hpp>
#include <boost/type_traits/add_const.hpp>

// Это нам понадобится на этапе 5.
#include <boost/mpl/transform.hpp>
```

2. Теперь давайте поместим всю магию метапрограммирования в структуру для более простого повторного использования:

```
template <class Types, class Modifiers>
struct do_modifications {
```

3. Рекомендуется проверить, что переданные векторы имеют одинаковый размер:

```
BOOST_STATIC_ASSERT((boost::is_same<
    typename boost::mpl::size<Types>::type,
    typename boost::mpl::size<Modifiers>::type
>::value));
```

4. Теперь давайте позаботимся о модифицирующей метафункции:

```
typedef boost::mpl::if_<
    boost::is_same<boost::mpl::_2, unsigne>,
    boost::make_unsigned<boost::mpl::_1>,
    boost::mpl::if_<
        boost::is_same<boost::mpl::_2, constant>,
        boost::add_const<boost::mpl::_1>,
        boost::mpl::_1
    >
> binary_operator_t;
```

5. И последний шаг:

```
    typedef typename boost::mpl::transform<
        Types,
        Modifiers,
        binary_operator_t
    >::type type;
};
```

Теперь мы запустим несколько тестов и убедимся, что наша метафункция отлично работает:

```
#include <boost/mpl/vector.hpp>
#include <boost/mpl/at.hpp>

typedef boost::mpl::vector<
    unsigne, no_change, constant, unsigne
> modifiers;

typedef boost::mpl::vector<
    int, char, short, long
> types;

typedef do_modifications<types, modifiers>::type result_type;

BOOST_STATIC_ASSERT((boost::is_same<
    boost::mpl::at_c<result_type, 0>::type,
    unsigned int
>::value));

BOOST_STATIC_ASSERT((boost::is_same<
    boost::mpl::at_c<result_type, 1>::type,
    char
>::value));

BOOST_STATIC_ASSERT((boost::is_same<
    boost::mpl::at_c<result_type, 2>::type,
    const short
>::value));

BOOST_STATIC_ASSERT((boost::is_same<
    boost::mpl::at_c<result_type, 3>::type,
    unsigned long
>::value));
```

Как это работает...

На этапе 3 мы проверяем, что размеры равны, но мы делаем это новым способом. Метафункция `boost::mpl::size<Types>::type` возвращает целочисленную константу `boost::mpl::long_<4>`, поэтому в проверке времени компиляции мы фактически сравниваем два типа, а не два числа. Это можно переписать более привычным способом:

```
BOOST_STATIC_ASSERT((
    boost::mpl::size<Types>::type::value
    ==
    boost::mpl::size<Modifiers>::type::value
));
```

Обратите внимание на ключевое слово `typename`, которое мы используем. Без него компилятор не сможет понять, является ли `::type` на самом деле типом или некой переменной. Предыдущие рецепты не требовали этого, потому что параметры для метафункции были полностью известны в той точке, где мы их использовали. Но в этом рецепте параметр для метафункции – это шаблон.

Мы рассмотрим этап 5, прежде чем позаботиться об этапе 4. На этапе 5 мы передаем параметры `Types`, `Modifiers` и `binary_operator_t` из этапа 4 в метафункцию `boost::mpl::transform`. Это довольно простая метафункция – из каждого переданного вектора она берет элемент и передает его третьему параметру – метафункции. Если переписать это в псевдокод, то он будет выглядеть следующим образом:

```
void boost_mpl_transform_pseoudo_code() {
    vector result;
    for (std::size_t i = 0; i < Types.size(); ++i) {
        result.push_back(
            binary_operator_t(Types[i], Modifiers[i])
        );
    }
    return result;
}
```

Этап 4 у кого-то может вызвать головную боль. На этом этапе мы пишем метафункцию, которая вызывается для каждой пары типов из векторов `Types` и `Modifiers` (*см.* предыдущий псевдокод):

```
typedef boost::mpl::if_<
    boost::is_same<boost::mpl::_2, unsigne>,
    boost::make_unsigned<boost::mpl::_1>,
    boost::mpl::if_<
        boost::is_same<boost::mpl::_2, constant>,
        boost::add_const<boost::mpl::_1>,
        boost::mpl::_1
    >
> binary_operator_t;
```

Как мы уже знаем, `boost::mpl::_2` и `boost::mpl::_1` – это заполнители. В этом рецепте `_1` является заполнителем для типа из вектора `Types`, а `_2` – это заполнитель для типа из вектора `Modifiers`.

Итак, вот как работает вся метафункция.

1. Она сравнивает второй передаваемый ей параметр (через _2) с типом `unsigne`.
2. Если типы равны, она делает первый передаваемый ей через `_1` параметр беззнаковым и возвращает этот тип.
3. В противном случае она сравнивает второй передаваемый ей через `_2` параметр с типом `constant`.
4. Если типы равны, она делает первый передаваемый ей через `_1` параметр константным и возвращает этот тип.
5. В противном случае она возвращает первый передаваемый ей через `_1` параметр.

Мы должны быть осторожны при создании этой метафункции. Не стоит в конце вызывать `::type`:

```
>::type binary_operator_t; // INCORRECT!
```

Если мы вызываем `::type`, компилятор попытается вычислить наш бинарный оператор в этой точке, и это приведет к ошибке компиляции. В псевдокоде такая попытка будет выглядеть так:

```
binary_operator_t foo;
// Попытка вызвать binary_operator_t::operator() без параметров,
// когда имеется только перегрузка для двух параметров.
foo();
```

Дополнительно...

Работа с метафункциями требует практики. Даже ваш покорный слуга не может правильно написать некоторые функции с первой попытки (вторая и третья попытки тоже не всегда удачны). Не бойтесь и не стесняйтесь экспериментировать!

Библиотека `Boost.MPL` не является частью C++ и не использует современные возможности C++, но ее можно применять с вариадическими шаблонами C++11:

```
template <class... T>
struct vt_example {
    typedef typename boost::mpl::vector<T...> type;
};

BOOST_STATIC_ASSERT((boost::is_same<
    boost::mpl::at_c<vt_example<int, char, short>::type, 0>::type,
    int
>::value));
```

Как и всегда, метафункции не добавляют ни одной инструкции к получающемуся двоичному файлу и не ухудшают производительность. Однако, используя их, вы подстраиваете свой код для работы с разными типами.

См. также

- Прочтите эту главу с самого начала, чтобы увидеть более простые примеры использования библиотеки `Boost.MPL`;
- *см.* главу 4 «*Уловки времени компиляции*», в особенности рецепт «*Выбор оптимального оператора для шаблонного параметра*», который содержит код, похожий на метафункцию `binary_operator_t`;
- официальная документация по `Boost.MPL` содержит еще больше примеров и полное оглавление: http://boost.org/libs/mpl.

Получение результирующего типа функции во время компиляции

В C++11 было добавлено много хороших возможностей для упрощения метапрограммирования. Одной из таких возможностей является альтернативный синтаксис функции. Он позволяет проще выводить результирующий тип шаблонной функции. Вот пример:

```
template <class T1, class T2>
auto my_function_cpp11(const T1& v1, const T2& v2)
    -> decltype(v1 + v2)
{
    return v1 + v2;
}
```

Это позволяет нам легче писать универсальные функции:

```
#include <cassert>

struct s1 {};
struct s2 {};
struct s3 {};

inline s3 operator + (const s1& /*v1*/, const s2& /*v2*/) {
    return s3();
}

inline s3 operator + (const s2& /*v1*/, const s1& /*v2*/) {
    return s3();
}

int main() {
    s1 v1;
    s2 v2;

    s3 res0 = my_function_cpp11(v1, v2);
    assert(my_function_cpp11('\0', 1) == 1);
}
```

Но в Boost есть множество шаблонных функций, и для его работы не требуется C++11. Как такое возможно и как сделать версию функции `my_function_cpp11` для C++03?

Подготовка

Для этого рецепта требуются базовые знания C++ и шаблонов.

Как это делается...

C++11 значительно упрощает метапрограммирование. На C++03 придется написать много кода, чтобы получить результат, похожий на альтернативный синтаксис функций.

1. Мы должны подключить следующий заголовочный файл:

```
#include <boost/type_traits/common_type.hpp>
```

2. Теперь давайте создадим метафункцию в пространстве имен `result_of` для любых типов:

```
namespace result_of {

    template <class T1, class T2>
    struct my_function_cpp03 {
        typedef typename boost::common_type<T1, T2>::type type;
    };
```

3. И специализируем ее для типов `s1` и `s2`:

```
    template <>
    struct my_function_cpp03<s1, s2> {
        typedef s3 type;
    };
    template <>
    struct my_function_cpp03<s2, s1> {
        typedef s3 type;
    };
} // Пространство имен result_of
```

4. Теперь мы готовы написать функцию `my_function_cpp03`:

```
template <class T1, class T2>
typename result_of::my_function_cpp03<T1, T2>::type
    my_function_cpp03(const T1& v1, const T2& v2)
{
    return v1 + v2;
}
```

Готово!

```
int main() {
    s1 v1;
    s2 v2;

    s3 res1 = my_function_cpp03(v1, v2);
    assert(my_function_cpp03('\0', 1) == 1);
}
```

Как это работает...

Основная идея этого рецепта заключается в том, что мы можем создать специальную метафункцию, которая выводит результирующий тип. Такой метод можно увидеть во всех библиотеках Boost, например в реализации `boost::get<>` библиотеки `Boost.Variant` или почти в любой функции библиотеки `Boost.Fusion`.

Теперь давайте двигаться шаг за шагом. Пространство имен `result_of` – своего рода традиция, но вы можете использовать собственное пространство имен, это не столь важно. Метафункция `boost::common_type<>` выводит тип, общий для нескольких типов, поэтому мы используем ее для общего случая. Мы также добавили две шаблонные специализации структур `result_of::my_function_cpp03` для типов `s1` и `s2`.

Недостаток написания метафункций в C++03 состоит в том, что иногда нам приходится много писать. Чтобы почувствовать разницу, сравните количество кода для `my_function_cpp11` и `my_function_cpp03`, включая пространство имен `result_of`.

Когда метафункция готова, мы можем вывести результирующий тип без C++11:

```
template <class T1, class T2>
typename result_of::my_function_cpp03<T1, T2>::type
    my_function_cpp03(const T1& v1, const T2& v2);
```

Дополнительно...

Этот метод не добавляет расходов времени выполнения, но может немного замедлить компиляцию. Вы также можете использовать его на современных компиляторах C++.

См. также

- Рецепты главы 4 «*Уловки времени компиляции*» предоставят вам гораздо больше информации о `Boost.TypeTraits` и метапрограммировании;
- ознакомьтесь с официальной документацией по `Boost.TypeTraits` для получения дополнительной информации о готовых метафункциях на странице http://boost.org/libs/type_traits.

Создание метафункции высшего порядка

Функции, которые принимают другие функции в качестве входного параметра, или функции, которые возвращают другие функции, называют **функциями высшего порядка**. Например:

```
typedef void(*function_t)(int);

function_t higher_order_function1();
void higher_order_function2(function_t f);
function_t higher_order_function3(function_t f); f);
```

Мы уже встречали метафункции высшего порядка в рецептах «*Использование типа “вектор типов”*» и «*Манипулирование вектором типов*» данной главы, где работали с `boost::mpl::transform`.

В этом рецепте мы попытаемся создать собственную метафункцию высшего порядка с именем `coalesce`, которая принимает два типа и две метафункции. Метафункция `coalesce` будет применять первый параметр типа к первой метафункции и сравнивать результирующий тип с типом `boost::mpl::false_`. Если результирующий тип является типом `boost::mpl::false_`, функция возвращает результат применения второго параметра типа ко второй метафункции, в противном случае она возвращает первый результирующий тип. В псевдокоде эта функция выглядит так:

```
template <class Param1, class Param2, class Func1, class Func2>
struct coalesce {
    using result1 = Func1(Param1);
    using type = (typeid(result1) == boost::mpl::false_ ? Func2(Param2) : result1);
};
```

Подготовка

Этот рецепт (и глава) непрост. Настоятельно рекомендуется читать данную главу с самого начала.

Как это делается...

Метафункции `Boost.MPL` на самом деле являются структурами, которые можно с легкостью передать в качестве параметра шаблона. Трудность состоит в том, чтобы правильно использовать его.

1. Чтобы написать метафункцию высшего порядка, нам нужны следующие заголовочные файлы:

```
#include <boost/mpl/apply.hpp>
#include <boost/mpl/if.hpp>
#include <boost/type_traits/is_same.hpp>
```

2. Следующий шаг – вычисление наших функций:

```
template <class Param1, class Param2, class Func1, class Func2>
struct coalesce {
    typedef typename boost::mpl::apply<Func1, Param1>::type type1;
    typedef typename boost::mpl::apply<Func2, Param2>::type type2;
```

3. Теперь нам нужно выбрать правильный результирующий тип:

```
    typedef typename boost::mpl::if_<
        boost::is_same< boost::mpl::false_, type1>,
        type2,
        type1
    >::type type;
};
```

Готово! Мы создали метафункцию высшего порядка! Теперь можно использовать ее, как показано ниже:

```
#include <boost/static_assert.hpp>
#include <boost/mpl/not.hpp>
#include <boost/mpl/next.hpp>

using boost::mpl::_1;
using boost::mpl::_2;

typedef coalesce<
    boost::mpl::true_,
    boost::mpl::int_<5>,
    boost::mpl::not_<_1>,
    boost::mpl::next<_1>
>::type res1_t;
BOOST_STATIC_ASSERT((res1_t::value == 6));

typedef coalesce<
    boost::mpl::false_,
    boost::mpl::int_<5>,
    boost::mpl::not_<_1>,
    boost::mpl::next<_1>
>::type res2_t;
BOOST_STATIC_ASSERT((res2_t::value));
```

Как это работает...

Основная проблема при написании метафункций высшего порядка – заполнители. Чтобы они работали, мы не должны вызывать `Func1<Param1>::type` напрямую. Вместо этого мы должны использовать метафункцию `boost::mpl::apply`, которая принимает одну функцию и до пяти параметров, которые должны быть переданы этой функции.

Можно настроить `boost::mpl::apply`, чтобы принимать еще больше параметров, определив макрос `BOOST_MPL_LIMIT_METAFUNCTION_ARITY` для требуемого количества параметров.

Дополнительно...

C++11 не имеет ничего похожего на функционал `Boost.MPL` для применения мета-функции.

В современном C++ есть множество функций, которые могут помочь вам достичь функциональности `Boost.MPL`. Например, в C++11 есть заголовочный файл `<type_traits>` и поддержка **базового спецификатора constexpr**. В C++14 имеется поддержка **расширенного спецификатора constexpr**, в C++17 есть функция `std::apply`, которая работает с кортежами и может использоваться в константных выражениях. Также в C++17 лямбда-выражения по умолчанию являются constexpr, и есть **if constexpr**.

На написание собственного решения с использованием C++11 ушло бы много времени. Таким образом, библиотеки, подобные `Boost.MPL`, по-прежнему остаются одним из наиболее подходящих решений для метапрограммирования.

См. также

См. официальную документацию, особенно раздел *Tutorial*, для получения дополнительной информации о `Boost.MPL` на странице http://boost.org/libs/mpl.

Ленивое вычисление метафункции

Ленивое вычисление означает, что функция не вызывается, пока нам действительно не понадобится ее результат. Знание этого рецепта настоятельно рекомендуется для написания хороших метафункций. Важность ленивых вычислений будет показана в следующем примере.

Представьте, что мы пишем некую метафункцию, которая принимает функцию `Func`, параметр `Param` и условие `Cond`. Результирующий тип этой функции должен быть типом `fallback`, если при применении `Cond` к `Param` возвращается значение `false`, в противном случае результатом должна быть `Func`, примененная к `Param`:

```
struct fallback;

template <
        class Func,
        class Param,
        class Cond,
        class Fallback = fallback>
struct apply_if;
```

Эта метафункция – то место, где мы не можем жить без ленивых вычислений, потому что может быть невозможно применить `Func` к `Param`, если `Cond` не соблюдается. Такая попытка приводит к сбою компиляции, и `Fallback` так и не возвращается.

Подготовка

Настоятельно рекомендуется прочитать главу 4 «*Уловки времени компиляции*». Однако хорошего знания метапрограммирования должно быть достаточно.

Как это делается...

Следите за мелкими деталями, например за тем, где мы не вызываем `::type`.

1. Нам понадобятся следующие заголовочные файлы:

```
#include <boost/mpl/apply.hpp>
#include <boost/mpl/eval_if.hpp>
#include <boost/mpl/identity.hpp>
```

2. Начало у функции простое:

```
template <class Func, class Param, class Cond, class Fallback>
struct apply_if {
    typedef typename boost::mpl::apply<
        Cond, Param
    >::type condition_t;
```

3. Здесь мы будем осторожны и не будем сразу выполнять метод:

```
    typedef boost::mpl::apply<Func, Param> applied_type;
```

4. При вычислении выражения следует проявлять дополнительную осторожность:

```
    typedef typename boost::mpl::eval_if_c<
            condition_t::value,
            applied_type,
            boost::mpl::identity<Fallback>
        >::type type;
};
```

Готово! Теперь мы можем использовать наш код следующим образом:

```
#include <boost/static_assert.hpp>
#include <boost/type_traits/is_integral.hpp>
#include <boost/type_traits/make_unsigned.hpp>
#include <boost/type_traits/is_same.hpp>

using boost::mpl::_1;
using boost::mpl::_2;

typedef apply_if<
    boost::make_unsigned<_1>,
    int,
    boost::is_integral<_1>
>::type res1_t;

BOOST_STATIC_ASSERT((
    boost::is_same<res1_t, unsigned int>::value
));

typedef apply_if<
    boost::make_unsigned<_1>,
    float,
    boost::is_integral<_1>
>::type res2_t;

BOOST_STATIC_ASSERT((
    boost::is_same<res2_t, fallback>::value
));
```

Как это работает...

Основная идея этого рецепта состоит в том, что мы не должны выполнять метафункцию, если условие ложно, потому что в этом случае есть вероятность, что метафункцию для этого типа применить нельзя:

```
// Провалится проверка времени компиляции где-то глубоко в реализации
// of boost::make_unsigned<_1>, если мы не воспользуемся ленивыми вычислениями
typedef apply_if<
    boost::make_unsigned<_1>,
    float,
    boost::is_integral<_1>
>::type res2_t;

BOOST_STATIC_ASSERT((
    boost::is_same<res2_t, fallback>::value
));
```

Итак, почему же у нас получилась ленивая метафункция?
Компилятор не заглядывает внутрь метафункции, если нет доступа к внутренним типам или значениям метафункции. Другими словами, компилятор пытается скомпилировать метафункцию, только когда мы пытаемся получить один из ее членов через `::`. Это может быть вызовом `::type` или `::value`. Вот как выглядит неправильная версия `apply_if`:

```
template <class Func, class Param, class Cond, class Fallback>
struct apply_if {
    typedef typename boost::mpl::apply<
        Cond, Param
    >::type condition_t;

   // Неверно: метафункция вычисляется сразу из-за `::type`
   typedef typename boost::mpl::apply<Func, Param>::type applied_type;

   typedef typename boost::mpl::if_c<
       condition_t::value,
       applied_type,
       boost::mpl::identity<Fallback>
   >::type type;
};
```

Это отличается от нашего примера, где мы не вызывали `::type` на *этапе 3* и реализовали *этап 4*, используя `eval_if_c`, который вызывает `::type` только для одного из своих параметров. Метафункция `boost::mpl::eval_if_c` реализуется так:

```
template<bool C, typename F1, typename F2>
struct eval_if_c {
    typedef typename if_c<C,F1,F2>::type f_;
    typedef typename f_::type type; // `::type` вызывается только для одного из параметров
};
```

Поскольку `boost::mpl::eval_if_c` вызывает `::type` для успешного условия, а у `fallback` нет `::type`, мы были обязаны обернуть `fallback` в класс `boost::mpl::identity`. Этот класс является очень простой, но полезной структурой, которая возвращает свой параметр шаблона через вызов `::type` и больше ничего не делает:

```
template <class T>
struct identity {
    typedef T type;
};
```

Дополнительно...

Как мы уже упоминали, в C++ практически нет классов `Boost.MPL`, но мы можем использовать `std::common_type<T>` с одним аргументом для получения функционала `boost::mpl::identity<T>`. В C++20 появился `std::type_identity<T>`, работающий как `boost::mpl::identity<T>`.

Как и всегда, метафункции не добавляют накладных расходов времени выполнения, и вы можете использовать метафункции столько раз, сколько захотите. Чем больше вы делаете во время компиляции, тем меньше остается работы для времени выполнения.

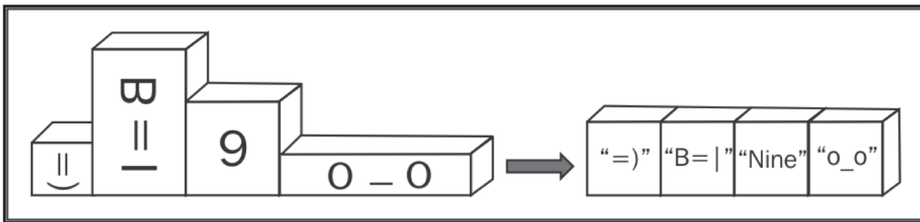
См. также

- Тип `boost::mpl::identity` можно использовать для отключения **поиска, зависящего от аргументов** (англ. **Argument-dependent lookup, ADL**) в шаблонных функциях. *См.* исходники `boost::implicit_cast` в заголовочном файле `<boost/implicit_cast.hpp>`;
- может быть полезно прочитать эту главу с самого начала и официальную документацию по `Boost.MPL` по адресу http://boost.org/libs/mpl.

Преобразование всех элементов кортежа в строки

Этот и последующий рецепты посвящены сочетанию возможностей времени компиляции и выполнения. Мы будем использовать библиотеку `Boost.Fusion` и посмотрим, что она может сделать.

Помните, что мы говорили о кортежах и массивах в первой главе? Теперь мы хотим написать одну функцию, которая преобразовывает каждый из элементов кортежа или массива в строки.



Подготовка

Вы должны быть знакомы с классами `boost::tuple` и `boost::array` и функцией `boost::lexical_cast`.

Как это делается...

Мы уже знаем почти все функции и классы, которые будут использоваться в этом рецепте. Просто нужно собрать их всех вместе.

1. Нам нужно написать функтор, который преобразовывает любой тип в строку:

```
#include <boost/lexical_cast.hpp>
#include <boost/noncopyable.hpp>

struct stringize_functor {
private:
    std::string& result;

public:
    explicit stringize_functor(std::string& res)
        : result(res)
    {}

    template <class T>
    void operator()(const T& v) const {
        result += boost::lexical_cast<std::string>(v);
    }
};
```

2. А теперь самая сложная часть кода:

```
#include <boost/fusion/include/for_each.hpp>

template <class Sequence>
std::string stringize(const Sequence& seq) {
    std::string result;
    boost::fusion::for_each(seq, stringize_functor(result));
    return result;
}
```

Вот и все! Сейчас можно преобразовать все, что угодно, в строку:

```
#include <iostream>
#include <boost/fusion/include/vector.hpp>
#include <boost/fusion/adapted/boost_tuple.hpp>
#include <boost/fusion/adapted/std_pair.hpp>
#include <boost/fusion/adapted/boost_array.hpp>

struct cat{};

std::ostream& operator << (std::ostream& os, const cat& ) {
    return os << "Meow! ";
}

int main() {
    boost::fusion::vector<cat, int, std::string> tup1(cat(), 0, "_0");
    boost::tuple<cat, int, std::string> tup2(cat(), 0, "_0");
    std::pair<cat, cat> cats;
    boost::array<cat, 10> many_cats;

    std::cout << stringize(tup1) << '\n'
        << stringize(tup2) << '\n'
        << stringize(cats) << '\n'
        << stringize(many_cats) << '\n';
}
```

В предыдущем примере выводится это:

```
Meow! 0_0
Meow! 0_0
Meow! Meow!
Meow! Meow! Meow! Meow! Meow! Meow! Meow! Meow! Meow! Meow!
```

Как это работает...

Основная проблема при написании функции stringize заключается в том, что ни boost::tuple, ни std::pair не имеют методов begin() или end(), поэтому мы не можем вызвать std::for_each. Здесь в дело вступает Boost.Fusion.

Библиотека Boost.Fusion содержит множество потрясающих алгоритмов, которые могут манипулировать структурами во время компиляции.

Функция boost::fusion::for_each перебирает элементы последовательности и применяет функтор для каждого из элементов.

Обратите внимание, что мы подключили следующие заголовочные файлы:

```
#include <boost/fusion/adapted/boost_tuple.hpp>
#include <boost/fusion/adapted/std_pair.hpp>
#include <boost/fusion/adapted/boost_array.hpp>
```

Это необходимо, потому что по умолчанию Boost.Fusion работает только с собственными классами. У Boost.Fusion есть собственный класс кортежей boost::fusion::vector, который очень похож на boost::tuple:

```
#include <string>
#include <cassert>

#include <boost/tuple/tuple.hpp>

#include <boost/fusion/include/vector.hpp>
#include <boost/fusion/include/at_c.hpp>

void tuple_example() {
    boost::tuple<int, int, std::string> tup(1, 2, "Meow");
    assert(boost::get<0>(tup) == 1);
    assert(boost::get<2>(tup) == "Meow");
}

void fusion_tuple_example() {
    boost::fusion::vector<int, int, std::string> tup(1, 2, "Meow");
    assert(boost::fusion::at_c<0>(tup) == 1);
    assert(boost::fusion::at_c<2>(tup) == "Meow");
}
```

Но класс boost::fusion::vector не так прост, как boost::tuple. Мы увидим разницу между ними в рецепте «*Расщепление кортежей*».

Дополнительно...

Существует одно фундаментальное различие между boost::fusion::for_each и std::for_each. Функция std::for_each содержит внутри себя цикл и определяет во время выполнения, сколько итераций должно быть сделано. Однако

`boost::fusion::for_each()` знает количество итераций во время компиляции и полностью разворачивает цикл. Для `boost::tuple<cat, int, std::string> tup2` вызов `boost::fusion::for_each(tup2, functor)` эквивалентен следующему коду:

```
functor(boost::fusion::at_c<0>(tup2));
functor(boost::fusion::at_c<1>(tup2));
functor(boost::fusion::at_c<2>(tup2));
```

В C++11 нет классов `Boost.Fusion`. Все методы этой библиотеки очень эффективны. Они делают как можно больше вычислений во время компиляции и имеют очень продвинутые оптимизации.

C++14 добавляет `std::integer_sequence` и `std::make_integer_sequence`. Используя эти сущности, можно вручную написать функциональность `boost::fusion::for_each` и реализовать функцию `stringize` без `Boost.Fusion`:

```
#include <utility>
#include <tuple>

template <class Tuple, class Func, std::size_t... I>
void stringize_cpp11_impl(const Tuple& t, const Func& f, std::index_sequence<I...>) {
   // Ой. Требуются выражения свертки (англ. Fold expressions) C++17.
   // (f(std::get<I>(t)), ...);

   int tmp[] = { 0, (f(std::get<I>(t)), 0)... };
   (void)tmp; // Подавление предупреждения о неиспользуемой переменной.
}

template <class Tuple>
std::string stringize_cpp11(const Tuple& t) {
    std::string result;
    stringize_cpp11_impl(
        t,
        stringize_functor(result),
        std::make_index_sequence< std::tuple_size<Tuple>::value >()
    );
    return result;
}
```

Как видите, было написано много кода, и такой код не так-то прост для чтения и понимания.

Идеи по добавлению чего-то похожего на `constexpr for` в стандарт C++23 обсуждаются в рабочей группе по стандартизации C++. С помощью такого функционала мы могли бы упростить прошлый код (синтаксис может измениться!):

```
template <class Tuple>
std::string stringize_cpp20(const Tuple& t) {
    std::string result;
    for constexpr(const auto& v: t) {
        result += boost::lexical_cast<std::string>(v);
    }
    return result;
}
```

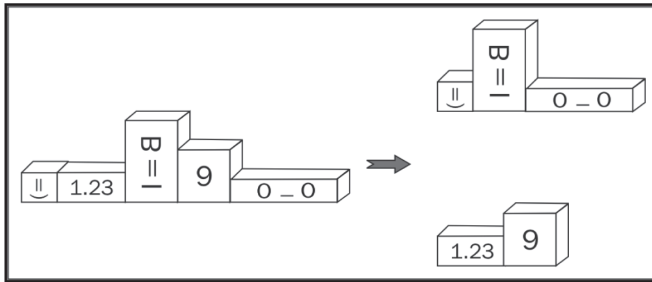
А до тех пор `Boost.Fusion` представляется наиболее переносимым и простым решением.

См. также

- Рецепт «Расщепление кортежей» даст больше информации об истинной мощи `Boost.Fusion`;
- несколько интересных примеров и полную справочную документацию можно найти по адресу http://boost.org/libs/fusion.

Расщепление кортежей

Этот рецепт покажет маленький кусочек возможностей библиотеки `Boost.Fusion`. Мы будем разбивать один кортеж на два: первый с арифметическими типами, а второй со всеми остальными.



Подготовка

Этот рецепт требует знания `Boost.MPL`, заполнителей и `Boost.Tuple`. Рекомендуется читать эту главу с самого начала.

Как это делается...

Это, возможно, один из самых сложных рецептов данной главы. Результирующие типы определяются во время компиляции, а значения для этих типов заполняются во время выполнения.

1. Нам нужны следующие заголовочные файлы:

```
#include <boost/fusion/include/remove_if.hpp>
#include <boost/type_traits/is_arithmetic.hpp>
```

2. Теперь мы готовы создать функцию, которая возвращает неарифметические типы:

```
template <class Sequence>
typename boost::fusion::result_of::remove_if<
    const Sequence,
    boost::is_arithmetic<boost::mpl::_1>
>::type get_nonarithmetics(const Sequence& seq)
{
    return boost::fusion::remove_if<
        boost::is_arithmetic<boost::mpl::_1>
```

```
        >(seq);
    }
```

3. И функцию, которая возвращает арифметические типы:

```
template <class Sequence>
    typename boost::fusion::result_of::remove_if<
        const Sequence,
        boost::mpl::not_< boost::is_arithmetic<boost::mpl::_1> >
    >::type get_arithmetics(const Sequence& seq)
    {
        return boost::fusion::remove_if<
            boost::mpl::not_< boost::is_arithmetic<boost::mpl::_1> >
        >(seq);
    }
```

Готово! Теперь мы можем написать следующий тестовый код:

```
#include <boost/fusion/include/vector.hpp>
#include <cassert>
#include <boost/fusion/include/at_c.hpp>
#include <boost/blank.hpp>

int main() {
    typedef boost::fusion::vector<
        int, boost::blank, boost::blank, float
    > tup1_t;
    tup1_t tup1(8, boost::blank(), boost::blank(), 0.0);

    boost::fusion::vector<boost::blank, boost::blank> res_na
        = get_nonarithmetics(tup1);
    boost::fusion::vector<int, float> res_a = get_arithmetics(tup1);
    assert(boost::fusion::at_c<0>(res_a) == 8);
}
```

Как это работает...

Идея `Boost.Fusion` заключается в том, что компилятор знает разметку структуры во время компиляции. А что известно компилятору во время компиляции, мы можем менять во время компиляции. `Boost.Fusion` позволяет нам изменять различные последовательности, добавлять и удалять поля, а также изменять типы полей. Это то, что мы сделали на *этапах 2* и *3*, – мы удалили ненужные поля из кортежа.

Теперь давайте внимательно рассмотрим `get_nonarithmetics`. Прежде всего его тип результата выводится с использованием следующей конструкции:

```
typename boost::fusion::result_of::remove_if<
    const Sequence,
    boost::is_arithmetic<boost::mpl::_1>
>::type
```

Это должно быть знакомо нам. Нечто подобное мы уже видели в рецепте «Получение типа результата функции во время компиляции». Заполнитель `Boost.MPL` `boost::mpl::_1` хорошо работает с метафункцией `boost::fusion::result_of::remove_if`, которая возвращает новый тип последовательности.

Теперь давайте переместимся внутрь функции и посмотрим на следующий код:

```
return boost::fusion::remove_if<
    boost::is_arithmetic<boost::mpl::_1>
>(seq);
```

Помните, что компилятор знает все типы внутри seq во время компиляции. Это означает, что Boost.Fusion может применять метафункции для различных элементов seq и получать для них результаты метафункции. Это также означает, что Boost.Fusion знает, как скопировать нужные поля из старой структуры в новую.

Однако Boost.Fusion старается не копировать поля как можно дольше.

Код на *этапе 3* очень похож на код на *этапе 2*, но у него есть дополнительный инвертирующий предикат для удаления типов, не попадающих в категорию арифметических.

Наши функции могут использоваться с любым типом, поддерживаемым Boost.Fusion, а не только с boost::fusion::vector.

Дополнительно…

Вы можете использовать функции Boost.MPL для контейнеров Boost.Fusion. Просто нужно подключить заголовок <boost/fusion/include/mpl.hpp>:

```
#include <boost/fusion/include/mpl.hpp>
#include <boost/mpl/transform.hpp>
#include <boost/type_traits/remove_const.hpp>

template <class Sequence>
struct make_nonconst: boost::mpl::transform<
    Sequence,
    boost::remove_const<boost::mpl::_1>
> {};

typedef boost::fusion::vector<
    const int, const boost::blank, boost::blank
> type1;
typedef make_nonconst<type1>::type nc_type;

BOOST_STATIC_ASSERT((boost::is_same<
    boost::fusion::result_of::value_at_c<nc_type, 0>::type,
    int
>::value));

BOOST_STATIC_ASSERT((boost::is_same<
    boost::fusion::result_of::value_at_c<nc_type, 1>::type,
    boost::blank
>::value));
```

```
BOOST_STATIC_ASSERT((boost::is_same<
    boost::fusion::result_of::value_at_c<nc_type, 2>::type,
    boost::blank
>::value));
```

Мы использовали `boost::fusion::result_of::value_at_c` вместо `boost::fusion::result_of::at_c`, потому что `boost::fusion::result_of::at_c` возвращает точный тип возврата вызова `boost::fusion::at_c`, который является ссылкой, а `boost::fusion::result_of::value_at_c` возвращает тип без ссылки.

Библиотеки `Boost.Fusion` и `Boost.MPL` не являются частью стандарта C++. `Boost.Fusion` – очень быстрая библиотека, и у нее имеется множество оптимизаций.

Стоит отметить, что мы увидели только крошечную часть способностей `Boost.Fusion`. О ней можно написать отдельную книгу.

См. также

- Неплохие пошаговые руководства и полная документация по `Boost.Fusion` доступны по адресу http://boost.org/libs/fusion;
- вы также можете просмотреть официальную документацию по `Boost.MPL` по адресу http://boost.org/libs/mpl.

Манипулирование гетерогенными контейнерами в C++14

Большинство приемов метапрограммирования, которые мы видели в этой главе, были придуманы задолго до появления C++11. Возможно, вы уже слышали о некоторых из них.

Как насчет чего-то нового? Как насчет реализации предыдущего рецепта в C++14 с помощью библиотеки, которая переворачивает метапрограммирование с ног на голову, отчего у вас поднимутся брови? Пристегните ремни безопасности, мы погружаемся в мир `Boost.Hana`.

Подготовка

Этот рецепт требует знания C++11 и C++14, в особенности лямбда-выражений. Вам понадобится совместимый с C++14 компилятор.

Как это делается...

Давайте теперь делать все в стиле `Boost.Hana`.

1. Начнем с подключения заголовочного файла:

```
#include <boost/hana/traits.hpp>
```

2. Мы создаем функциональный объект `is_arithmetic_`:

```
constexpr auto is_arithmetic_ = [](const auto& v) {
    auto type = boost::hana::typeid_(v);
    return boost::hana::traits::is_arithmetic(type);
};
```

3. Теперь мы реализуем функцию get_nonarithmetics:

```
#include <boost/hana/remove_if.hpp>

template <class Sequence>
auto get_nonarithmetics(const Sequence& seq) {
    return boost::hana::remove_if(seq, [](const auto& v) {
        return is_arithmetic_(v);
    });
}
```

4. Давайте определим get_arithmetics несколько другим способом, чем get_nonarithmetics. Просто ради удовольствия!

```
#include <boost/hana/filter.hpp>

constexpr auto get_arithmetics = [](const auto& seq) {
    return boost::hana::filter(seq, is_arithmetic_);
};
```

Вот и все. Теперь можно использовать эти функции:

```
#include <boost/hana/tuple.hpp>
#include <boost/hana/integral_constant.hpp>
#include <boost/hana/equal.hpp>
#include <cassert>

struct foo {
    bool operator==(const foo&) const { return true; }
    bool operator!=(const foo&) const { return false; }
};

int main() {
    const auto tup1
        = boost::hana::make_tuple(8, foo{}, foo{}, 0.0);

    const auto res_na = get_nonarithmetics(tup1);
    const auto res_a = get_arithmetics(tup1);

    using boost::hana::literals::operator ""_c;
    assert(res_a[0_c] == 8);

    const auto res_na_expected = boost::hana::make_tuple(foo(), foo());
    assert(res_na == res_na_expected);
}
```

Как это работает...

На первый взгляд код может показаться простым, но это не так. Boost.Hana переворачивает метапрограммирование! В предыдущих рецептах мы работали с типами напрямую, а Boost.Hana создает переменную и большую часть времени работает с типом переменной.

Посмотрите на вызов typeid_ на *этапе 2*:

```
auto type = boost::hana::typeid_(v);
```

Фактически здесь мы возвращаем переменную. Информация о типе теперь скрыта внутри переменной type. Ее можно извлечь, вызвав decltype(type)::type.

Но давайте двигаться построчно. На *этапе 2* мы сохраняем обобщенное лямбда-выражение в переменную is_arithmetic_. С этого момента мы можем использовать эту переменную в качестве функционального объекта. Внутри лямбда-выражения мы создаем переменную type, которая теперь содержит информацию о типе переменной v. Следующая строка – это специальная обертка вокруг std::is_arithmetic, которая извлекает информацию о типе переменной v из переменной type и передает ее в std::is_arithmetic. Результатом этого вызова является булева интегральная константа.

А теперь магия! Лямбда-выражение, хранящееся в переменной is_arithmetic_, фактически никогда не вызывается функциями boost::hana::remove_if и boost::hana::filter. Всем функциям Boost.Hana, которые его используют, нужен только тип результата лямбда-функции, но не ее тело. Мы можем смело изменить определение, и весь пример по-прежнему будет прекрасно работать:

```
constexpr auto is_arithmetic_ = [] (const auto& v) {
    assert(false);
    auto type = boost::hana::typeid_(v);
    return boost::hana::traits::is_arithmetic(type);
};
```

На *этапах 3* и *4* мы вызываем функции boost::hana::remove_if и boost::hana::filter соответственно. На *этапе 3* мы использовали переменную is_arithmetic_ внутри лямбда-выражения. На *этапе 4* мы использовали ее напрямую. Вы можете использовать любой синтаксис, какой пожелаете, это всего лишь дело привычки.

Наконец, в функции main() мы проверяем, что все работает как положено и что элемент в кортеже по индексу 0 равен 8:

```
using boost::hana::literals::operator ""_c;
assert(res_a[0_c] == 8);
```

Лучший способ понять библиотеку Boost.Hana – поэкспериментировать с ней. Это можно сделать онлайн на странице http://apolukhin.github.io/Boost-Cookbook/.

Дополнительно...

Есть небольшая деталь, оставленная неописанной. Как работает доступ к кортежу через operator[]? Невозможно иметь одну функцию, которая возвращает разные типы!

operator ""_c из библиотеки Boost.Hana работает с литералами и создает различные типы в зависимости от литерала:

- если написать 0_c, вернется integral_constant<long long, 0>;
- если написать 1_c, вернется integral_constant<long long, 1>;
- если написать 2_c, вернется integral_constant<long long, 2>.

Класс boost::hana::tuple на самом деле имеет много перегрузок operator[], принимающих различные типы integral_constant. В зависимости от значения

интегральной константы возвращается правильный элемент кортежа. Например, если вы напишете `some_tuple[1_c]`, то вызывается `tuple::operator[](integra_constant<long long, 1>)` и возвращается элемент по индексу 1.

Библиотека `Boost.Hana` не является частью стандарта С++. Тем не менее ее автор участвует в собраниях по стандартизации С++ и предлагает различные интересные вещи для включения в стандарт С++.

Если вы ожидаете от `Boost.Hana` времени компиляции на порядок лучше, чем от `Boost.MPL`, то не стоит. В настоящее время компиляторы не очень хорошо поддерживают подход с использованием `Boost.Hana`. Однажды, возможно, что-то изменится.

Стоит посмотреть исходные коды библиотеки `Boost.Hana`, чтобы открыть для себя новые интересные способы использования возможностей С++14. Все библиотеки Boost можно найти на GitHub на странице https://github.com/boostorg.

См. также

Официальная документация содержит дополнительные примеры, обучающие материалы и раздел, посвященный производительности во время компиляции. Наслаждайтесь библиотекой `Boost.Hana` по адресу http://boost.org/libs/hana.

Глава 9

Контейнеры

Темы, которые мы рассмотрим в этой главе:

- хранение нескольких элементов в контейнере;
- хранение не более *N* элементов в контейнере;
- сверхбыстрое сравнение строк;
- использование неупорядоченных ассоциативных контейнеров;
- создание ассоциативного контейнера, с индексированием и по значениям;
- использование многоиндексных контейнеров;
- получение преимуществ от односвязного списка и пула памяти;
- использование плоских ассоциативных контейнеров.

Вступление

Эта глава посвящена контейнерам Boost и вещам, непосредственно связанным с ними. Она предоставляет информацию о классах Boost, которые можно использовать в повседневном программировании, что значительно ускорит ваш код и облегчит разработку новых приложений.

Контейнеры отличаются не только функциональностью, но и вычислительной сложностью некоторых своих членов. Знание сложностей имеет важное значение для написания быстрых приложений. Эта глава не просто знакомит вас с новыми контейнерами, она дает вам подсказки относительно того, когда следует и когда не следует использовать контейнер определенного типа или его методы.

Итак, давайте начнем!

Хранение нескольких элементов в контейнере

Последние два десятилетия программисты C++ по умолчанию использовали `std::vector` в качестве контейнера для хранения последовательностей. Это быстрый контейнер, который не выполняет большого количества динамических выделений, хранит элементы в кеш-дружелюбной манере, и поскольку контейнер хранит элементы непрерывно, функции, подобные `std::vector::data()`, позволяют взаимодействовать с функциями языка программирования C.

Но мы хотим большего! Есть случаи, когда мы знаем, какое типичное количество элементов хранится в векторе, и нам нужно улучшить производительность вектора, полностью исключив выделение памяти для этого случая.

Представьте, что мы пишем высокопроизводительную систему обработки банковских транзакций. **Транзакция** – это последовательность операций, все операции должны успешно выполниться, или все они должны отмениться, если по крайней мере одна из операций не удалась. Мы знаем, что 99 % транзакций состоят из восьми или менее операций, и мы хотим ускорить функцию:

```
#include <vector>

class operation;

template <class T>
void execute_operations(const T&);

bool has_operation();
operation get_operation();

void process_transaction_1() {
    std::vector<operation> ops;
    ops.reserve(8); // TODO: Выделение памяти. Медленно!

    while (has_operation()) {
        ops.push_back(get_operation());
    }

    execute_operations(ops);
    // ...
}
```

Подготовка

Этот рецепт требует только базовых знаний стандартной библиотеки и C++.

Как это делается...

Это будет самая простая задача из данной книги благодаря библиотеке `Boost.Container`.

1. Подключите соответствующий заголовочный файл:

```
#include <boost/container/small_vector.hpp>
```

2. Замените `std::vector` на `boost::container::small_vector` и удалите вызов функции `reserve()`:

```
void process_transaction_2() {
    boost::container::small_vector<operation, 8> ops;

    while (has_operation()) {
        ops.push_back(get_operation());
    }

    execute_operations(ops);
    // ...
}
```

Как это работает...

Второй шаблонный параметр класса boost::container::small_vector – это количество элементов для предварительного выделения в стеке. Поэтому, если большую часть времени нам нужно хранить восемь или менее элементов в векторе, мы просто помещаем 8 в качестве второго параметра шаблона.

Если нам нужно хранить более восьми элементов в контейнере boost::container::small_vector<operation, 8>, то small_vector ведет себя точно так же, как std::vector, и динамически выделяет фрагмент памяти для хранения более восьми элементов. Как и std::vector, small_vector является **контейнером последовательностей** (англ. **sequence container**) с **итераторами произвольного доступа** (англ. **random access iterator**), который последовательно хранит элементы.

Подводя итог, следует сказать, что boost::container::small_vector – это контейнер, который ведет себя точно так же, как std::vector, но позволяет избежать выделения памяти для указанного во время компиляции количества элементов.

Дополнительно...

Недостаток использования small_vector состоит в том, что предположение о количестве элементов просачивается в сигнатуру функции, которая принимает small_vector в качестве параметра. Поэтому если у нас есть три разных small_vector для 4, 8 и 16 элементов соответственно, и то нам понадобится три разные функции execute_operations, чтобы с ними работать:

```
void execute_operations(
    const boost::container::small_vector<operation, 4>&);

void execute_operations(
    const boost::container::small_vector<operation, 8>&);

void execute_operations(
  const boost::container::small_vector<operation, 16>&);
```

Это плохо! Теперь у нас есть несколько функций в нашем исполняемом файле, которые делают одно и то же и состоят из почти одинаковых машинных кодов. Это приводит к увеличению размера двоичных файлов, увеличению времени запуска исполняемого файла, увеличению времени компиляции и времени компоновки. Некоторые компиляторы могут устранить эту избыточность, но шансы на это невелики.

Однако есть очень простое решение. boost::container::small_vector публично наследуется от типа boost::container::small_vector_base, который не зависит от количества элементов, но обладает всеми функциями-членами vector:

```
void execute_operations(
    const boost::container::small_vector_base<operation>& ops
);
```

Готово! Теперь мы можем использовать новую функцию execute_operations с любым boost::container::small_vector, не боясь увеличения размера двоичного файла, лишних копирований и динамических аллокаций.

В C++20 нет класса, подобного small_vector. Есть предложения включить small_vector в следующий стандарт C++, который появится где-то в 2023 году.

См. также

- Библиотека `Boost.Container` содержит полную справочную документацию по многим интересным классам по адресу http://boost.org/libs/container;
- `small_vector` пришел в Boost из проекта **LLVM**; вы можете прочитать об этом контейнере на сайте http://llvm.org/docs/ProgrammersManual.html#llvm-adt-smallvector-h.

Хранение не более *N* элементов в контейнере

Вот вопрос: какой контейнер мы должны использовать для возврата последовательности из функции, если знаем, что последовательность никогда не имеет более *N* элементов, а *N* – небольшое число. Например, как мы должны написать функцию `get_events()`, которая возвращает максимум пять событий:

```
#include <vector>
std::vector<event> get_events();
```

`std::vector<event>` динамически выделяет память, поэтому такой код – плохое решение.

```
#include <boost/array.hpp>
boost::array<event, 5> get_events();
```

`boost::array<event, 5>` не выделяет память, но создает все пять элементов. Нельзя вернуть менее пяти элементов.

```
#include <boost/container/small_vector.hpp>
boost::container::small_vector<event, 5> get_events();
```

`boost::container::small_vector<event, 5>` не выделяет память для пяти или менее элементов и позволяет вернуть менее пяти элементов. Однако это решение не идеально, поскольку из интерфейса функции не очевидно, что она никогда не возвращает более пяти элементов.

Подготовка

Этот рецепт требует только базовых знаний стандартной библиотеки и C++.

Как это делается...

В библиотеке `Boost.Container` есть контейнер, который полностью удовлетворяет нашим потребностям:

```
#include <boost/container/static_vector.hpp>
boost::container::static_vector<event, 5> get_events();
```

Как это работает...

`boost::container::static_vector<T, N>` – это контейнер, который может содержать не более указанного во время компиляции количества элементов. Рассматривайте его как `boost::container::small_vector<T, N>`, который просто не может динамически выделять память, и любая попытка сохранить более *N* элементов приводит к исключению `std::bad_alloc`:

```
#include <cassert>

int main () {
    boost::container::static_vector<event, 5> ev = get_events();
    assert(ev.size() == 5);

    boost::container::static_vector<int, 2> ints;
    ints.push_back(1);
    ints.push_back(2);
    try {
        // Приведенная ниже строка всегда выбрасывает исключение:
        ints.push_back(3);
    } catch (const std::bad_alloc& ) {
        // ...
    }
}
```

Как и все контейнеры библиотеки `Boost.Container`, `static_vector` поддерживает **семантику перемещений** и эмулирует rvalue-ссылки с помощью библиотеки `Boost.Move`, если ваш компилятор не поддерживает rvalue.

Дополнительно...

Если пользователь вставляет элемент в `std::vector` и при этом невозможно вставить новое значение в уже выделенную память, то `std::vector` выделит больший кусок памяти. В этом случае `std::vector` перемещает элементы из старого местоположения в новое, если они являются `std::is_nothrow_move_constructible`. В противном случае `std::vector` копирует элементы в новое местоположение и после этого вызывает деструктор для каждого из них в старом местоположении.

Из-за этого поведения `std::vector` имеет **амортизированную константную сложность O(1)** для многих функций-членов. `static_vector` никогда не выделяет память и не перемещает элементы из старого местоположения в новое. Из-за этого операции, которые имеют амортизированную сложность O(1) для `std::vector`, имеют истинную сложность O(1) для `boost::container::static_vector`. Это может быть удобно для некоторых приложений, работающих в режиме реального времени; однако остерегайтесь исключений!

Некоторые по-прежнему предпочитают передавать выходные параметры по ссылке, а не возвращать их: `void get_events(static_vector<event, 5>& result_out)`. Они думают, что таким образом есть гарантия, что копирования результата не происходит. Не делайте этого, так будет только хуже! Компиляторы C++ имеют целый ряд оптимизаций, таких как **оптимизация возвращаемого значения (RVO) и оптимизация именованного возвращаемого значения (NRVO)**; у разных платформ имеются соглашения ABI, где закреплено, что код с `retun что-либо` не приводит к появлению ненужной копии и т. д. Никакого копирования уже не происходит. Однако когда вы передаете значение по ссылке, компилятор просто не видит, откуда взято значение, и может предположить, что несколько ссылок могут ссылаться на одно и то же значение. Это называется алиасинг (англ. aliasing), и он может значительно ухудшить производительность.

В C++17 нет класса static_vector, и на данный момент не планируется добавлять его в C++23.

См. также

Официальная документация по Boost.Container содержит подробный справочный раздел, в котором описаны все функции-члены класса boost::container::static_vector. Перейдите по ссылке http://boost.org/libs/container.

Сверхбыстрое сравнение строк

Манипулирование строками – распространенная задача. Здесь мы увидим, как можно быстро выполнить операцию сравнения строк, используя несколько простых приемов. Этот рецепт является трамплином для следующего рецепта, где методы, описанные здесь, будут использоваться для достижения константного времени поиска.

Итак, нам нужно создать некий класс, способный быстро сравнивать строки на равенство значений. Мы создадим шаблонную функцию для измерения скорости сравнения:

```
#include <string>

template <class T>
std::size_t test_default() {
    // Константы
    const std::size_t ii_max = 200000;
    const std::string s(
        "Long long long string that "
        "will be used in tests to compare "
        "speed of equality comparisons."
    );
    // Создаем некие данные, которые будут использоваться в сравнениях.
    const T data1[] = {
       T(s),
       T(s + s),
       T(s + ". Whooohooo"),
       T(std::string(""))
    };

    const T data2[] = {
       T(s),
       T(s + s),
       T(s + ". Whooohooo"),
       T(std::string(""))
    };

  const std::size_t data_dimensions = sizeof(data1) / sizeof(data1[0]);

  std::size_t matches = 0u;
  for (std::size_t ii = 0; ii < ii_max; ++ii) {
      for (std::size_t i = 0; i < data_dimensions; ++i) {
```

```
            for (std::size_t j = 0; j < data_dimensions; ++j) {
                if (data1[i] == data2[j]) {
                    ++ matches;
                }
            }
        }
    }

    return matches;
}
```

Подготовка

Этот рецепт требует только базовых знаний стандартной библиотеки и C++.

Как это делается...

Мы сделаем `std::string` публичным полем в нашем собственном классе и добавим для него операторы сравнения. Методы для работы с полем `std::string` для краткости примера писать не будем.

1. Нам понадобится следующий заголовочный файл:

```
#include <boost/functional/hash.hpp>
```

2. Теперь мы можем создать наш класс для быстрых сравнений:

```
struct string_hash_fast {
    typedef std::size_t comp_type;

    const comp_type comparison_;
    const std::string str_;

    explicit string_hash_fast(const std::string& s)
        : comparison_(
            boost::hash<std::string>()(s)

        )
        , str_(s)
     {}
};
```

3. Не забудьте определить операторы сравнения:

```
inline bool operator == (
    const string_hash_fast& s1, const string_hash_fast& s2)
{
    return s1.comparison_ == s2.comparison_ && s1.str_ == s2.str_;
}
inline bool operator != (
    const string_hash_fast& s1, const string_hash_fast& s2)
{
    return !(s1 == s2);
}
```

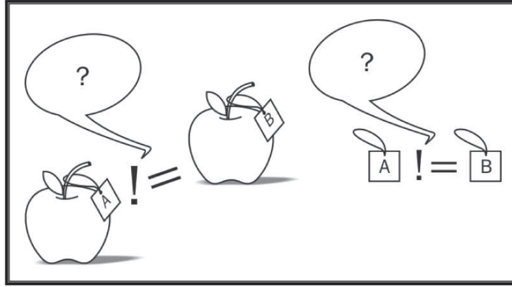
4. Вот и все! Теперь мы можем запустить наши тесты и увидеть результат, используя этот код:

```
#include <iostream>
#include <iostream>
#include <cassert>

int main(int argc, char* argv[]) {
    if (argc < 2) {
        assert(
            test_default<string_hash_fast>()
            ==
            test_default<std::string>()
        );
        return 0;
    }

    switch (argv[1][0]) {
    case 'h':
        std::cout << "HASH matched: "
                  << test_default<string_hash_fast>();
        break;

    case 's':
        std::cout << "STD matched: "
                  << test_default<std::string>();
        break;

    default:
        return 2;
    }
}
```

Как это работает...

Сравнение строк происходит медленно, потому что если строки имеют одинаковую длину, мы должны сравнивать все символы строки по одному. Вместо этого в примере мы заменяем сравнение строк на сравнение целых чисел. Это делается с помощью функции `hash` – функции, которая создает короткое представление строки с фиксированной длиной.

Давайте поговорим о значениях `hash` на примере яблок. Представьте, что у вас есть два яблока с этикетками, как показано на приведенной ниже диаграмме, и вы хотите проверить, что яблоки одного сорта. Самый простой способ сравнить эти яблоки – сравнить их по этикеткам. В противном случае вы потеряете много времени, сравнивая яблоки по цвету, размеру, форме и другим параметрам. Хеш – это что-то вроде этикетки, которая отражает значение объекта.



Теперь давайте будем двигаться шаг за шагом.

На *этапе 1* мы подключаем заголовочный файл, который содержит определения функций `hash`. На *этапе 2* мы объявляем наш новый класс строки, содержащий `str_`, который является исходным значением строки, и поле `comparison_`, которое является вычисляемым хеш-значением. Обратите внимание на конструкцию:

```
boost::hash<std::string>()(s)
```

Здесь `boost::hash<std::string>` – это структура, функциональный объект, такой же, как и `std::negate<>`. Вот почему нам нужна первая скобка – мы создаем этот функциональный объект. Вторая скобка с `s` внутри – это вызов `std::size_t operator()(const std::string& s)`, который вычисляет значение хеша.

Теперь посмотрим на *этап 3*, где мы определяем `operator==`:

```
return s1.comparison_ == s2.comparison_ && s1.str_ == s2.str_;
```

Внимание на вторую часть выражения. Операция хеширования теряет информацию, а это означает, что, возможно, есть несколько разных строк, которые выдают одно и то же хеш-значение. На практике это означает, что если хеши не совпадают, существует 100%-ная гарантия, что строки не совпадают; в противном случае мы должны сравнивать строки, используя традиционные методы.

Ну что же, пора сравнивать числа. Если мы измерим время выполнения, используя метод сравнения по умолчанию, то получим 819 миллисекунд; однако наше сравнение работает почти в два раза быстрее и завершается за 475 миллисекунд.

Дополнительно...

В C++11 есть функциональный объект `hash`; его можно найти в заголовке `<functional>` в пространстве имен `std::`. Хеширование в Boost и стандартной библиотеке работает быстро. Оно не выделяет динамически память, а также не имеет виртуальных функций.

Вы можете специализировать хеширование для своих типов. В Boost это делается с помощью написания функции `hash_value` в пространстве имен пользовательского типа:

```
// Должно находиться в пространстве имен класса string_hash_fast.
inline std::size_t hash_value(const string_hash_fast& v) {
    return v.comparison_;
}
```

Этот подход отличается от подхода стандартной библиотеки, где для настройки поведения `std::hash` требуется написать специализацию структуры `hash<>` в пространстве имен `std::`.

Хеширование в Boost определено для всех основных типов (таких как `int`, `float`, `double` и `char`), для массивов и контейнеров всех стандартных библиотек, включая `std::array`, `std::tuple` и `std::type_index`. Некоторые библиотеки также предоставляют хеш-специализации, например библиотека `Boost.Variant` может хешировать любые классы `boost::variant`.

См. также

- Прочтите рецепт «*Использование неупорядоченных ассоциативных контейнеров*» этой главы для получения более подробной информации о применении хеш-функций;
- официальная документация по `Boost.Functional` и `Boost.Hash` расскажет вам, как объединить несколько хешей, и предоставит дополнительные примеры: http://boost.org/libs/functional/hash.

Использование неупорядоченных ассоциативных контейнеров

В предыдущем рецепте мы видели, как можно оптимизировать сравнение строк с помощью хеширования. После прочтения этого рецепта у вас, скорее всего, возникнет вопрос: можно ли создать контейнер, который будет кешировать хешированные значения, чтобы использовать более быстрое сравнение?

Ответ – да, и это еще не все. Мы можем добиться почти постоянного времени поиска, вставки и удаления элементов.

Подготовка

Требуются базовые знания контейнеров C++ и STL. Чтение предыдущего рецепта также будет полезно.

Как это делается...

Просто.

1. Все, что вам нужно сделать, – это просто подключить заголовочный файл `<boost/unordered_map.hpp>`, если вы хотите использовать неупорядоченные ассоциативные контейнеры с ключами и значениями. Если вам нужны только ключи и не нужны значения, подключите заголовочный файл `<boost/unordered_set.hpp>`.
2. Теперь вы можете использовать `boost::unordered_map` вместо `std::map` и `boost::unordered_set` вместо `std::set`:

```
#include <boost/unordered_set.hpp>
#include <string>
#include <cassert>

void example() {
    boost::unordered_set<std::string> strings;
```

```
        strings.insert("This");
        strings.insert("is");
        strings.insert("an");
        strings.insert("example");

        assert(strings.find("is") != strings.cend());
    }
```

Как это работает...

Неупорядоченные контейнеры хранят значения и запоминают хеш каждого значения. Теперь, если вы хотите найти в контейнере значение, он вычислит хеш этого значения и найдет его в контейнере. После того как хеш найден, контейнер проверит найденное значение и искомое значение на равенство. Затем возвращается итератор на значение или на конец контейнера.

Поскольку контейнер может искать интегральное хеш-значение, он может использовать оптимизации и алгоритмы, подходящие только для целых чисел. Эти алгоритмы гарантируют постоянную сложность поиска O(1), когда традиционные `std::set` и `std::map` обеспечивают худшую сложность O(log (*N*)), где *N* – количество элементов в контейнере. Все это значит, что чем больше элементов в традиционных `std::set` или `std::map`, тем медленнее они работают. А вот производительность неупорядоченных контейнеров не зависит от количества элементов.

Такая хорошая производительность никогда не дается даром. В неупорядоченных контейнерах значения неупорядочены (вы не удивлены, не так ли?). Это означает, что если мы выведем все элементы контейнера, то результат будет разным в зависимости от библиотеки и может не повторяться от запуска к запуску:

```
template <class T>
void output_example() {
    T strings;
    strings.insert("CZ");
    strings.insert("CD");
    strings.insert("A");
    strings.insert("B");

    std::copy(
        strings.begin(),
        strings.end(),
        std::ostream_iterator<std::string>(std::cout, " ")
    );
}
```

Для `std::set` и `boost::unordered_set` мы получим следующий вывод:

```
boost::unordered_set<std::string> : B A CD CZ
std::set<std::string> : A B CD CZ
```

Итак, насколько отличается производительность? Обычно это зависит от качества реализации. У меня получились следующие цифры:

```
For 100 elements:
Boost: map is 1.69954 slower than unordered map
Std: map is 1.54316 slower than unordered map
```

```
For 1000 elements:
Boost: map is 4.13714 slower than unordered map
Std: map is 2.12495 slower than unordered map

For 10000 elements:
Boost: map is 2.04475 slower than unordered map
Std: map is 2.23285 slower than unordered map

For 100000 elements:
Boost: map is 1.67128 slower than unordered map
Std: map is 1.68169 slower than unordered map
```

Производительность была измерена с использованием приведенного ниже блока кода:

```
T map;

for (std::size_t ii = 0; ii < ii_max; ++ii) {
    map[s + boost::lexical_cast<std::string>(ii)] = ii;
}

// Утверждение.
for (std::size_t ii = 0; ii < ii_max; ++ii) {
    assert(map[s + boost::lexical_cast<std::string>(ii)] == ii);
}
```

Этот код содержит создание множества временных строк, поэтому он не на 100 % правильно измеряет ускорение. Он приводится здесь лишь для того, чтобы показать, что неупорядоченные контейнеры обычно быстрее, чем упорядоченные.

Иногда может возникнуть задача, когда нам нужно использовать пользовательский тип в неупорядоченных контейнерах:

```
struct my_type {
    int val1_;
    std::string val2_;
};
```

Для этого нам нужно написать оператор сравнения для этого типа:

```
inline bool operator == (const my_type& v1, const my_type& v2) {
    return v1.val1_ == v2.val1_ && v1.val2_ == v2.val2_;
}
```

Нам также нужно специализировать функцию хеширования для этого типа. Если тип состоит из нескольких полей, обычно нам просто нужно объединить хеши всех полей, которые участвуют в сравнениях на равенство:

```
std::size_t hash_value(const my_type& v) {
    std::size_t ret = 0u;
    boost::hash_combine(ret, v.val1_);
    boost::hash_combine(ret, v.val2_);
    return ret;
}
```

Настоятельно рекомендуется объединять хеши, используя функцию boost::hash_combine.

Дополнительно...

Также доступны мультиверсии контейнеров: boost::unordered_multiset определен в заголовочном файле <boost/unordered_set.hpp>, а boost::unordered_multimap – в <boost/unordered_map.hpp>. Как и в стандартной библиотеке, мультиверсии контейнеров могут хранить несколько одинаковых значений ключа.

Все неупорядоченные контейнеры позволяют вам указывать собственный функтор для хеширования вместо стандартного boost::hash. Они также позволяют вам использовать собственный функтор сравнения вместо стандартного std::equal_to.

В C++11 есть все неупорядоченные контейнеры из библиотеки Boost. Их можно найти в заголовках <unordered_set> и <unordered_map>, в пространстве имен std::, вместо boost::. Версии Boost и стандартной библиотеки могут немного отличаться по производительности. Тем не менее неупорядоченные контейнеры Boost доступны даже на компиляторах C++03/C++98 и используют эмуляцию rvalue-ссылок через Boost.Move. Поэтому вы можете использовать эти контейнеры с некопируемыми, но перемещаемыми классами, даже на компиляторах для стандарта, предшествующего C++11.

В стандарте C++ нет функции hash_combine, поэтому вы должны написать свою:

```
template <class T>
inline void hash_combine(std::size_t& seed, const T& v)
{
    std::hash<T> hasher;
    seed ^= hasher(v) + 0x9e3779b9 + (seed<<6) + (seed>>2);
}
```

Или просто используйте boost::hash_combine.

Начиная с Boost версии 1.64 неупорядоченные контейнеры в Boost обладают функциональностью для извлечения и вставки узлов контейнера. Подобный функционал появился в стандартной библиотеке в C++17.

См. также

- Рецепт «*Использование эмуляции перемещения C++11*» главы 1 «*Приступаем к написанию приложения*» предоставит вам более подробную информацию об эмуляции rvalue-ссылок с использованием Boost.Move;
- более подробная информация о неупорядоченных контейнерах доступна на официальном сайте по адресу http://boost.org/libs/unordered;
- дополнительная информация о сочетании и вычислении хешей для диапазонов доступна по адресу http://boost.org/libs/functional/hash.

Создание ассоциативного контейнера с индексированием и по значениям

Несколько раз в году нам нужно что-то, что может хранить и индексировать по паре значений.

Нам нужно получить первую часть пары, используя вторую, и получить вторую часть, используя первую. Запутались?

Представьте, что мы создаем класс словарь. Когда пользователи помещают в него строки, класс должен возвращать идентификаторы, а когда пользователи помещают в него идентификаторы, класс должен возвращать строки.

Такой класс может пригодиться, например, для хранения логинов. Пользователи помещают логин в наш словарь и хотят получить уникальный идентификатор пользователя. Также хочется уметь получать все логины по идентификатору пользователя.

Посмотрим, как можно реализовать это с помощью Boost.

Подготовка

Для этого рецепта требуются базовые знания стандартной библиотеки и шаблонов.

Как это делается...

Этот рецепт посвящен способностям библиотеки `Boost.Bimap`. Давайте посмотрим, как можно использовать ее для реализации этой задачи.

1. Нам нужны следующие заголовочные файлы:

```
#include <iostream>
#include <boost/bimap.hpp>
#include <boost/bimap/multiset_of.hpp>
```

2. Теперь мы готовы описать структуру нашего словаря:

```
int main() {
    typedef boost::bimap<
        std::string,
        boost::bimaps::multiset_of<std::size_t>
    > name_id_type;

    name_id_type name_id;
```

3. Его можно заполнить, используя следующий синтаксис:

```
// Вставляем ключи <-> значения
name_id.insert(name_id_type::value_type(
    "John Snow", 1
));

name_id.insert(name_id_type::value_type(
    "Vasya Pupkin", 2
));
name_id.insert(name_id_type::value_type(
    "Antony Polukhin", 3
```

```
));
// Тот же человек, что и "Antony Polukhin"
name_id.insert(name_id_type::value_type(
   "Anton Polukhin", 3
));
```

4. Мы можем работать с левой частью как с ассоциативным контейнером:

```
std::cout << "Left:\n";

typedef name_id_type::left_const_iterator left_const_iterator;
const left_const_iterator lend = name_id.left.end();

for (left_const_iterator it = name_id.left.begin();
     it!= lend;
     ++it)
{
    std::cout << it->first << " <=> " << it->second << '\n';
}
```

5. Правая часть почти такая же, как левая:

```
std::cout << "\nRight:\n";

typedef name_id_type::right_const_iterator right_const_iterator;
const right_const_iterator rend = name_id.right.end();

for (right_const_iterator it = name_id.right.begin();
     it!= rend;
     ++it)
{
    std::cout << it->first << " <=> " << it->second << '\n';
}
```

6. Мы также можем убедиться, что в словаре есть такой человек:

```
   assert(name_id.left.find("John Snow")->second == 1);
   assert(name_id.right.find(2)->second == "Vasya Pupkin");
   assert(
       name_id.find(name_id_type::value_type(
        "Anton Polukhin", 3
       )) != name_id.end()
   );
} /* Конец функции main() */
```

Вот и все. Теперь, если мы поместим весь код (кроме заголовков) внутрь `int main()`, то получим следующий результат:

```
Left:
Anton Polukhin <=> 3
Antony Polukhin <=> 3
John Snow <=> 1
Vasya Pupkin <=> 2

Right:
1 <=> John Snow
```

```
2 <=> Vasya Pupkin
3 <=> Antony Polukhin
3 <=> Anton Polukhin
```

Как это работает...

На этапе 2 мы определяем тип `bimap`:

```
typedef boost::bimap<
    std::string,
    boost::bimaps::multiset_of<std::size_t>
> name_id_type;
```

Первый параметр шаблона сообщает, что первичный ключ должен иметь тип `std::string` и индексироваться как `std::set`. Второй параметр шаблона сообщает, что вторичный ключ должен иметь тип `std::size_t` и что несколько разных первичных ключей могут иметь одинаковое значение вторичного ключа, как в `std::multimap`.

Мы можем настраивать поведение `bimap`, используя классы из пространства имен `boost::bimaps::`. Мы можем использовать неупорядоченный контейнер для индексирования первого ключа:

```
#include <boost/bimap/unordered_set_of.hpp>
#include <boost/bimap/unordered_multiset_of.hpp>

typedef boost::bimap<
    boost::bimaps::unordered_set_of<std::string>,
    boost::bimaps::unordered_multiset_of<std::size_t>
> hash_name_id_type;
```

Когда мы не указываем способ индексирования ключа и просто указываем его тип, `Boost.Bimap` использует `boost::bimaps::set_of` в качестве индексирования по умолчанию. Мы можем взять наш пример:

```
#include <boost/bimap/set_of.hpp>

typedef boost::bimap<
    boost::bimaps::set_of<std::string>,
    boost::bimaps::multiset_of<std::size_t>
> name_id_type;
```

И попытаться реализовать его средствами стандартной библиотеки. Это будет выглядеть как комбинация следующих двух переменных:

```
std::map<std::string, std::size_t> key1; // == name_id.left
std::multimap<std::size_t, std::string> key2; // == name_id.right
```

Как видно из предыдущих комментариев, вызов `name_id.left` (на *этапе 4*) возвращает ссылку на нечто с интерфейсом, похожим на `std::map<std::string, std::size_t>`.

Вызов `name_id.right` из *этапа 5* возвращает нечто с интерфейсом, похожим на `std::multimap <std::size_t, std::string>`.

На *этапе 6* мы работаем с целым `bimap`, ищем пару ключей и следим, чтобы они были в контейнере.

Дополнительно...

К сожалению, в стандарте C++ нет ничего, что было бы похоже на `Boost.Bimap`. Вот еще одна плохая новость: `Boost.Bimap` не поддерживает rvalue-ссылки, и на некоторых компиляторах отображается безумное количество предупреждений. Обратитесь к документации своего компилятора, чтобы получить информацию о подавлении предупреждений.

Хорошая новость состоит в том, что обычно `Boost.Bimap` использует меньше памяти, чем два контейнера стандартной библиотеки, а поиск выполняет так же быстро. Внутри `Boost.Bimap` нет вызовов виртуальных функций, но эта библиотека использует динамическое выделение памяти.

См. также

- Следующий рецепт «*Использование многоиндексных контейнеров*» даст вам больше информации о библиотеке для мультииндексирования, которую можно использовать вместо `Boost.Bimap`;
- прочитайте официальную документацию, где приводятся дополнительные примеры и сведения о `bimap`: http://boost.org/libs/bimap.

Использование многоиндексных контейнеров

В предыдущем рецепте мы создали что-то вроде словаря, который неплохо себя показал, когда нам нужно работать с парами значений. Но что, если нам понадобится гораздо более продвинутая индексация? Давайте создадим программу, которая индексирует людей:

```
struct person {
    std::size_t id_;
    std::string name_;
    unsigned int height_;
    unsigned int weight_;

    person(std::size_t id, const std::string& name,
                unsigned int height, unsigned int weight)
        : id_(id)
        , name_(name)
        , height_(height)
        , weight_(weight)
    {}
};

inline bool operator < (const person& p1, const person& p2) {
    return p1.name_ < p2.name_;
}
```

Нам понадобится много индексов, например по имени, идентификатору, росту и весу.

Подготовка

Необходимы базовые знания о контейнерах стандартной библиотеки, в частности о неупорядоченных ассоциативных контейнерах.

Как это делается...

Создавать все индексы и управлять ими можно с помощью одного контейнера из `Boost.Multiindex`.

1. Для этого нам нужно подключить множество заголовочных файлов:

```
#include <iostream>
#include <boost/multi_index_container.hpp>
#include <boost/multi_index/ordered_index.hpp>
#include <boost/multi_index/hashed_index.hpp>
#include <boost/multi_index/identity.hpp>
#include <boost/multi_index/member.hpp>
```

2. Сложнее всего описать тип `multi-index`:

```
void example_main() {
    typedef boost::multi_index::multi_index_container<
        person,
        boost::multi_index::indexed_by<
            // Имена уникальны и упорядочены
            boost::multi_index::ordered_unique<
                boost::multi_index::identity<person>
            >,

            // Идентификаторы не уникальны, но нам не важен их порядок
            boost::multi_index::hashed_non_unique<
                boost::multi_index::member<
                    person, std::size_t, &person::id_
                >
            >,

           // Рост может быть неуникальным, но его нужно отсортировать
           boost::multi_index::ordered_non_unique<
               boost::multi_index::member<
                   person, unsigned int, &person::height_
               >
           >,

           // Вес может быть неуникальным, но его нужно отсортировать
           boost::multi_index::ordered_non_unique<
               boost::multi_index::member<
                   person, unsigned int, &person::weight_
               >
           >
        > // конец `boost::multi_index::indexed_by<`
    > indexes_t;
```

3. Теперь мы можем вставить значения в наш `multi_index`:

```
indexes_t persons;

// Вставка значений:
persons.insert(person(1, "John Snow", 185, 80));
persons.insert(person(2, "Vasya Pupkin", 165, 60));
persons.insert(person(3, "Antony Polukhin", 183, 70));
// Тот же человек, что и "Antony Polukhin". Но почему-то на 1 см ниже
persons.insert(person(3, "Anton Polukhin", 182, 70));
```

4. Давайте создадим функцию для вывода содержимого индекса на экран:

```
template <std::size_t IndexNo, class Indexes>
void print(const Indexes& persons) {
     std::cout << IndexNo << ":\n";

     typedef typename Indexes::template nth_index<
             IndexNo
     >::type::const_iterator const_iterator_t;

     for (const_iterator_t it = persons.template get<IndexNo>().begin(),
          iend = persons.template get<IndexNo>().end();
          it != iend;
          ++it)
     {
         const person& v = *it;
         std::cout
             << v.name_ << ", "
             << v.id_ << ", "
             << v.height_ << ", "
             << v.weight_ << '\n'
         ;
     }
     std::cout << '\n';
}
```

5. Выведем все индексы:

```
print<0>(persons);
print<1>(persons);
print<2>(persons);
print<3>(persons);
```

6. Также можно использовать код из предыдущего рецепта:

```
assert(persons.get<1>().find(2)->name_ == "Vasya Pupkin");
assert(
    persons.find(person(
        77, "Anton Polukhin", 0, 0
    )) != persons.end()
);

// Не скомпилируется:
//assert(persons.get<0>().find("John Snow")->id_ == 1);
```

Теперь, если мы запустим наш пример, будет выведено содержимое индексов:

```
0:
Anton Polukhin, 3, 182, 70
Antony Polukhin, 3, 183, 70
John Snow, 1, 185, 80
Vasya Pupkin, 2, 165, 60

1:
John Snow, 1, 185, 80
Vasya Pupkin, 2, 165, 60
Anton Polukhin, 3, 182, 70
Antony Polukhin, 3, 183, 70

2:
Vasya Pupkin, 2, 165, 60
Anton Polukhin, 3, 182, 70
Antony Polukhin, 3, 183, 70
John Snow, 1, 185, 80

3:
Vasya Pupkin, 2, 165, 60
Antony Polukhin, 3, 183, 70
Anton Polukhin, 3, 182, 70
John Snow, 1, 185, 80
```

Как это работает...

Самая сложная часть здесь – это построение многоиндексного типа с использованием `boost::multi_index::multi_index_container`. Первый параметр шаблона – это класс, который мы будем индексировать. В нашем случае это `person`. Второй параметр – тип `boost::multi_index::indexed_by`, все индексы должны быть описаны как параметр шаблона этого класса.

Теперь давайте посмотрим на первое описание индекса:

```
boost::multi_index::ordered_unique<
    boost::multi_index::identity<person>
>
```

Использование класса `boost::multi_index::order_unique` означает, что индекс должен работать как `std::set` и у него должны быть все его члены. Класс `boost::multi_index::identity<person>` означает, что в индексе должен использоваться `operator<` класса `person` для упорядочения.

В приведенной ниже таблице показана связь между типами `Boost.MultiIndex` и контейнерами стандартной библиотеки C++:

Типы Boost.MultiIndex	Контейнеры STL
boost::multi_index::ordered_unique	std::set
boost::multi_index::ordered_non_unique	std::multiset
boost::multi_index::hashed_unique	std::unordered_set
boost::multi_index::hashed_non_unique	std::unordered_mutiset
boost::multi_index::sequenced	std::list

Посмотрите на второй индекс:

```
boost::multi_index::hashed_non_unique<
    boost::multi_index::member<
        person, std::size_t, &person::id_
    >
>
```

Тип `boost::multi_index::hashed_non_unique` означает, что индекс работает как `std::unordered_multiset`, а `boost::multi_index::member<person, std::size_t, &person::id_>` означает, что хеш-функция должна применяться только к полю `person::id_`.

Остальные индексы теперь вы сможете понять самостоятельно.

Давайте посмотрим на использование индексов в функции `print`. Получение типа итератора для индекса выполняется с помощью следующего кода:

```
typedef typename Indexes::template nth_index<
        IndexNo
>::type::const_iterator const_iterator_t;
```

Выглядит немного сложнее, потому что `Indexes` – это параметр шаблона. Пример был бы проще, если бы можно было написать этот код в области видимости `indexes_t`, а не в шаблонной функции:

```
typedef indexes_t::nth_index<0>::type::const_iterator const_iterator_t;
```

Метафункция `nth_index` принимает порядковый номер индекса (отсчет от нуля). В нашем примере индекс 1 – это индекс идентификаторов, индекс 2 – индекс роста и т. д.

Теперь давайте посмотрим, как использовать `const_iterator_t`:

```
for (const_iterator_t it = persons.template get<IndexNo>().begin(),
         iend = persons.template get<IndexNo>().end();
         it != iend;
         ++it)
    {
        const person& v = *it;
        // ...
```

Это код также можно упростить, если `indexes_t` находится в области видимости:

```
for (const_iterator_t it = persons.get<0>().begin(),
     iend = persons.get<0>().end();
     it != iend;
     ++it)
{
    const person& v = *it;
    // ...
```

И можно упростить еще сильнее, если в проекте использовать C++11:

```
for (const person& v : persons.template get<0>())
{
    // ...
```

Функция `get<indexNo>()` возвращает индекс. Можно использовать этот индекс почти как контейнер стандартной библиотеки.

Дополнительно...

В C++ нет многоиндексной библиотеки. Boost.MultiIndex – это быстрая библиотека, которая не использует виртуальные функции. Официальная документация по Boost.MultiIndex содержит показатели производительности и использования памяти, библиотека зачастую использует меньше памяти, чем рукописный код на основе стандартной библиотеки. boost::multi_index::multi_index_container поддерживает возможности C++11 и эмуляцию rvalue-ссылок с использованием Boost.Move с версии Boost 1.68.

См. также

Официальная документация по Boost.MultiIndex содержит обучающие материалы, показатели производительности, примеры и другую полезную информацию по библиотеке Boost.Multiindex на странице http://boost.org/libs/multi_index.

Получение преимуществ от односвязного списка и пула памяти

В настоящее время мы обычно используем std::vector, когда нам нужны неассоциативные и неупорядоченные контейнеры. Это рекомендуют Андрей Александреску и Херб Саттер в своей книге *C++ Coding Standards*. Даже те пользователи, которые не читали эту книгу, обычно используют std::vector. Почему? Ну, std::list медленнее и применяет гораздо больше ресурсов по сравнению с std::vector. Контейнер std::deque очень похож на std::vector, но не хранит значения в памяти непрерывно.

Однако если нам нужен контейнер, в котором удаление и вставка элементов не делают итераторы недействительными, то мы вынуждены выбрать медленный std::list.

Но подождите, мы можем подобрать более подходящее решение, используя Boost!

Подготовка

Необходимы хорошие знания о контейнерах стандартной библиотеки для понимания вводной части. После этого требуются только базовые знания C++ и стандартных контейнеров.

Как это делается...

В этом рецепте мы будем использовать две библиотеки Boost одновременно: Boost.Pool и односвязный список из Boost.Container.

1. Нам нужны следующие заголовки:

```
#include <boost/pool/pool_alloc.hpp>
#include <boost/container/slist.hpp>
#include <cassert>
```

2. Теперь нам нужно описать тип нашего списка. Это можно сделать, как показано в приведенном ниже коде:

```
typedef boost::fast_pool_allocator<int> allocator_t;
typedef boost::container::slist<int, allocator_t> slist_t;
```

3. Мы можем работать с нашим списком, как с `std::list`:

```
template <class ListT>
void test_lists() {
    typedef ListT list_t;

    // Вставляем 1 000 000 нулей.
    list_t list(1000000, 0);

    for (int i = 0; i < 1000; ++i) {
        list.insert(list.begin(), i);
    }

    // Ищем значение.
    typedef typename list_t::iterator iterator;
    iterator it = std::find(list.begin(), list.end(), 777);
    assert(it != list.end());

    // Стираем значения.
    for (int i = 0; i < 100; ++i) {
        list.pop_front();
    }

    // Итератор по-прежнему действителен и указывает на то же значение.
    assert(it != list.end());
    assert(*it == 777);

    // Вставляем дополнительные значения.
    for (int i = -100; i < 10; ++i) {
        list.insert(list.begin(), i);
    }
    // Итератор по-прежнему действителен и указывает на то же значение.
    assert(it != list.end());
    assert(*it == 777);
}

void test_slist() {
    test_lists<slist_t>();
}

void test_list() {
    test_lists<std::list<int> >();
}
```

4. Некоторые специфичные для списка функции:

```
void list_specific(slist_t& list, slist_t::iterator it) {
    typedef slist_t::iterator iterator;

    // Стираем элемент 776
    assert( *(++iterator(it)) == 776);
    assert(*it == 777);

    list.erase_after(it);
```

```
        assert(*it == 777);
        assert( *(++iterator(it)) == 775);
```

5. Нужно освободить память с помощью этого кода:

```
    // Освобождаем память: slist вызывает rebind для allocator_t и выделяет
    // узлы (ноды, англ. nodes), а не просто int.

    boost::singleton_pool<
        boost::fast_pool_allocator_tag,
        sizeof(slist_t::stored_allocator_type::value_type)
    >::release_memory();
} // Конец функции list_specific
```

Как это работает...

Когда мы используем `std::list`, то память под каждый узел списка выделяется отдельно. Это означает, что когда мы вставляем 10 элементов в `std::list`, контейнер вызывает `new` 10 раз. Кроме того, выделенные узлы обычно расположены случайным образом в памяти, что не идет на пользу кешу ЦП.

Вот почему мы использовали `boost::fast_pool_allocator<int>` из `Boost.Pool`. Этот аллокатор пытается выделять большие блоки памяти, чтобы позднее можно было создать несколько узлов в блоке без вызовов `new`.

У библиотеки `Boost.Pool` есть недостаток – она использует память для внутренних нужд. Обычно используется дополнительный `sizeof(void*)` для каждого элемента. Чтобы обойти проблему дополнительного потребления памяти, мы применяем односвязный список из `Boost.Containers`.

Класс `boost::container::slist` более компактен, но его итераторы могут работать только в одном направлении. Этап 3 прост для тех читателей, которые знают о контейнерах стандартной библиотеки, поэтому мы переходим к этапу 4, чтобы увидеть особенности `boost::container::slist`. Поскольку итератор односвязного списка может выполнять итерацию только в одном направлении, традиционные алгоритмы вставки и удаления занимают линейное время O(*N*). Это связано с тем, что, когда мы стираем или вставляем элемент, предыдущий элемент списка должен быть изменен, но мы не можем его получить из итератора. Чтобы обойти эту проблему, в односвязном списке есть методы `erase_after` и `insert_after`, которые работают за константное время O(1). С помощью этих методов мы вставляем или удаляем элементы сразу после текущей позиции итератора.

Тем не менее стирание и вставка значений в начале односвязного списка выполняются быстро.

Внимательно посмотрите на этот код:

```
boost::singleton_pool<
    boost::fast_pool_allocator_tag,
    sizeof(slist_t::stored_allocator_type::value_type)
>::release_memory();
```

boost::fast_pool_allocator не освобождает память, поэтому мы должны сделать это вручную. Рецепт «*Делайте это при выходе из области видимости*» главы 2 «*Управление ресурсами*» может помочь в освобождении Boost.Pool.

Давайте сравним время выполнения нашего решения и std::list:

```
$ TIME="Runtime=%E RAM=%MKB" time ./07_slist_and_pool l
std::list: Runtime=0:00.08 RAM=34224KB

$ TIME="Runtime=%E RAM=%MKB" time ./07_slist_and_pool s
slist_t: Runtime=0:00.04 RAM=19640KB
```

Как видно, slist_t использует в половину меньше памяти и в два раза быстрее по сравнению с классом std::list.

Дополнительно...

В библиотеке Boost.Container есть другое решение, boost::container::stable_vector. Он допускает произвольный доступ к элементам, имеет итераторы произвольного доступа, не инвалидирует итераторы при удалениях и вставках. Но при этом у него есть множество недостатков std::list в производительности и использовании памяти.

В C++11 есть std::forward_list, который очень близок к boost::containers::slist, а также методы *_after, но нет метода size(). Версии односвязного списка C++11 и Boost имеют одинаковую производительность, и ни у одной из них нет виртуальных функций. Однако версия для Boost также может использоваться на компиляторах C++03, и она даже поддерживает эмуляцию rvalue-ссылок через Boost.Move.

boost::fast_pool_allocator отсутствует в C++. Тем не менее в стандарте C++17 есть решение получше! Заголовок <memory_resource> содержит полезные материалы для работы с полиморфным аллокатором, и там же вы можете найти std::pmr::synchronized_pool_resource, std::pmr::unsynchronized_pool_resource и std::pmr::monotonic_buffer_resource. Поэкспериментируйте с ними, чтобы добиться еще лучшей производительности.

Угадайте, почему boost::fast_pool_allocator не освобождает память? Потому, что в C++03 нет поддержки аллокаторов с состояниями, поэтому контейнеры не копируют и не хранят аллокаторы, что делает невозможным реализовать освобождающий память деструктор для boost::fast_pool_allocator.

См. также

- Официальная документация по Boost.Pool содержит дополнительные примеры и классы для работы с пулами памяти. Перейдите по ссылке http://boost.org/libs/pool;
- рецепт «*Использование плоских ассоциативных контейнеров*» познакомит вас с другими классами из Boost.Container. Вы также можете прочитать официальную документацию по адресу http://boost.org/libs/container, чтобы самостоятельно изучить эту библиотеку или получить полную справочную документацию по ее классам;

- *Vector vs List* и другие интересные темы от Бьёрна Страуструпа, изобретателя языка программирования C++, можно найти на сайте https://channel9.msdn.com/Events/GoingNative/GoingNative-2012/Keynote-Bjarne-Stroustrup-Cpp11-Style.

Использование плоских ассоциативных контейнеров

Прочитав предыдущий рецепт, некоторые из вас могут начать использовать быстрые разделители пулов повсюду, особенно для `std::set` и `std::map`. Ну что же, я не буду мешать вам делать это, но давайте по крайней мере посмотрим на альтернативу: плоские ассоциативные контейнеры. Эти контейнеры реализованы поверх традиционного векторного и хранят упорядоченные значения.

Подготовка

Требуются базовые знания ассоциативных контейнеров стандартных библиотек.

Как это делается...

Плоские контейнеры являются частью библиотеки `Boost.Container`. Мы уже видели, как использовать некоторые из ее контейнеров в предыдущих рецептах. В этом рецепте мы будем применять ассоциативный контейнер `flat_set`.

1. Нам нужно будет подключить только один заголовочный файл:

```
#include <boost/container/flat_set.hpp>
```

2. После этого мы можем создать плоский контейнер и поэкспериментировать с ним:

```
#include <algorithm>
#include <cassert>

int main() {
    boost::container::flat_set<int> set;
```

3. Резервируем место для элементов:

```
set.reserve(4096);
```

4. Заполняем контейнер:

```
for (int i = 0; i < 4000; ++i) {
    set.insert(i);
}
```

5. Теперь мы можем работать с ним так же, как и с `std::set`:

```
// 5.1
assert(set.lower_bound(500) - set.lower_bound(100) == 400);

// 5.2
set.erase(0);

// 5.3
set.erase(5000);
```

```
    // 5.4
    assert(std::lower_bound(set.cbegin(), set.cend(), 900000) == set.cend());

    // 5.5
    assert(
        set.lower_bound(100) + 400
        ==
        set.find(500)
    );
} // Конец функции main()
```

Как это работает...

Этапы 1 и *2* не представляют сложности, а вот *этап 3* требует внимания. Это один из самых важных этапов при работе с плоскими ассоциативными контейнерами и `std::vector`.

Класс `boost::container::flat_set` хранит свои значения упорядоченно в векторе, а это означает, что любая вставка или удаление элементов, которые не находятся в конце контейнера, занимает время O(*N*), как и в случае с `std::vector`. Это неизбежное зло. Но взамен мы получаем почти втрое меньшее использование памяти на элемент, более дружественное к кешу процессора хранение и итераторы с произвольным доступом. Посмотрите на *этап 5*, `5.1`, где мы считаем расстояние между двумя итераторами, полученными из `lower_bound`. Получение расстояния с плоским контейнером занимает постоянное время O(1), в то время как та же самая операция для итераторов `std::set` занимает линейное время O(*N*). В случае с `5.1` получение расстояния с использованием `std::set` будет в 400 раз медленнее, чем получение расстояния для плоских контейнеров.

Вернемся к *этапу 3*. Без резервирования памяти вставка элементов может быть в разы медленнее и менее эффективной. Когда мы резервируем необходимое место, класс `std::vector` сразу выделяет кусок памяти нужного размера, а затем размещает в нем элементы. Когда мы вставляем элементы без резервирования памяти, есть вероятность, что на предварительно выделенном фрагменте памяти не осталось свободного места, поэтому `std::vector` выделяет фрагмент памяти побольше. После этого он копирует или перемещает элементы из первого фрагмента во второй, удаляет элементы первого фрагмента и освобождает его. Только тогда происходит вставка. Такое копирование и освобождение могут происходить по нескольку раз при вставке множества элементов, что значительно снижает производительность кода.

Если вы знаете количество элементов, которое должен хранить std::vector или любой другой плоский контейнер, зарезервируйте место для этих элементов перед вставкой. Это ускоряет работу программы в большинстве случаев!

Этап 4 прост, здесь мы вставляем элементы. Обратите внимание, что мы вставляем упорядоченные элементы. Это не обязательно, но рекомендуется для ускорения вставки. Вставка элементов в конец `std::vector` намного дешевле, чем в середину или в начало.

Этапы 5, 5.2 и 5.3 мало чем отличаются, кроме как по скорости их выполнения. Правила удаления элементов в значительной степени такие же, как и правила их вставки.

Возможно, вам кажется, что я рассказываю простые вещи о контейнерах, но я видел некоторые очень популярные продукты, использующие возможности C++11, которые имеют безумное количество оптимизаций и неэффективно используют контейнеры стандартной библиотеки, в частности std::vector.

На *этапе 5,* `5.4`, показано, что функция `std::lower_bound` работает быстрее с `boost::container::flat_set`, чем с `std::set`, благодаря итераторам произвольного доступа.

Этап 5, `5.5`, также показывает преимущества итераторов произвольного доступа.

Мы не использовали здесь функцию std::find, потому что эта функция ищет за линейное время O(*N*), а функции-члены find работают за логарифмическое время O(log (*N*)).

Дополнительно...

Когда нужно использовать плоские контейнеры, а когда обычные? Это решать вам, но вот список отличий из официальной документации по `Boost.Container`, который поможет вам принять решение:

- более быстрый поиск, по сравнению со стандартными ассоциативными контейнерами;
- гораздо более быстрая итерация, по сравнению со стандартными ассоциативными контейнерами;
- меньшее потребление памяти для маленьких объектов (и для больших объектов, если используется `shrink_to_fit`);
- улучшена производительность кеша (данные хранятся в непрерывной памяти);
- нестабильные итераторы (итераторы становятся недействительными при вставке и удалении элементов);
- некопируемые и неперемещаемые типы значений нельзя сохранить в плоском контейнере;
- более низкая безопасность исключений по сравнению со стандартными ассоциативными контейнерами (конструкторы копирования/перемещения элементов контейнера могут выбрасывать исключение при перемещении элементов в операциях удалений и вставок);
- более медленная вставка и удаление по сравнению со стандартными ассоциативными контейнерами.

В C++20, к сожалению, нет плоских контейнеров. Плоские контейнеры Boost быстрые, у них множество оптимизаций, и они не используют виртуальные функции. Классы из `Boost.Containers` поддерживают эмуляцию rvalue-ссылок через `Boost.Move`, поэтому вы можете свободно использовать их даже на компи-

ляторах C++03.

См. также

- Обратитесь к рецепту «*Получение преимуществ односвязного списка и пула памяти*» для получения дополнительной информации о библиотеке `Boost.Container`;
- рецепт «*Использование эмуляции перемещения C++11*» главы 1 «*Приступаем к написанию приложения*» познакомит вас с основами эмуляции rvalue-ссылок для компиляторов, совместимых с C++03;
- официальная документация по `Boost.Container` содержит много полезной информации об этой библиотеке и полную справочную документацию по каждому классу: http://boost.org/libs/container.

Глава 10

Сбор информации о платформе и компиляторе

Темы, которые мы рассмотрим в этой главе:

- обнаружение ОС и компилятора;
- обнаружение поддержки 128-битных целых чисел;
- обнаружение и обход отключенной динамической идентификации типа данных;
- написание метафункций с использованием более простых методов;
- уменьшение размера кода и повышение производительности пользовательских типов в C++11;
- переносимый способ экспорта и импорта функций и классов;
- обнаружение версии Boost и получение новейших функций.

Вступление

Различные проекты и компании имеют разные требования к кодированию. Некоторые из них запрещают исключения или динамическую идентификацию типов данных, а кто-то запрещает использование C++11. Если вы хотите писать переносимый код, который можно использовать в самых разных проектах, эта глава для вас.

Хотите сделать свой код максимально быстрым и использовать новейшие функции C++? Вам определенно понадобится инструмент для обнаружения функций компилятора.

Некоторые компиляторы имеют уникальные возможности, которые могут значительно упростить вам жизнь. Если вы ориентируетесь на один компилятор, то можете сэкономить много времени и использовать эти функции. Не нужно реализовывать свои аналоги с нуля!

Эта глава посвящена различным вспомогательным макросам, используемым для обнаружения функций компилятора, платформы и Boost. Эти макросы широко используются в библиотеках Boost и необходимы для написания переносимого кода, способного работать с любыми флагами компилятора.

Обнаружение ОС и компилятора

Я предполагаю, что вы встречали кучу некрасивых макросов для обнаружения компилятора, на котором скомпилирован код. Нечто подобное является типичной практикой в языке C:

```
#include <something_that_defines_macros>
#if !defined(__clang__) \
    && !defined(__ICC) \
    && !defined(__INTEL_COMPILER) \
    && (defined(__GNUC__) || defined(__GNUG__))

// GCC specific

#endif
```

Выглядит как-то так себе. Попробуйте придумать хороший макрос для обнаружения компилятора GCC. Постарайтесь сделать использование этого макроса максимально коротким.

Взгляните на приведенный далее рецепт, чтобы подтвердить свою догадку.

Подготовка

Требуются только базовые знания C++.

Как это делается...

Рецепт прост и состоит из одного заголовочного файла и одного макроса.

1. Заголовочный файл:

   ```
   #include <boost/predef/compiler.h>
   ```

2. Макрос:

   ```
   #if BOOST_COMP_GNUC

   // GCC specific

   #endif
   ```

Как это работает...

Заголовок `<boost/remaf/compiler.h>` знает все возможные компиляторы, и у него имеется макрос для каждого из них. Таким образом, если текущий компилятор – это компилятор GCC, то макрос `BOOST_COMP_GNUC` определен как `1`, а все остальные макросы для других компиляторов определены как `0`. Если это не компилятор GCC, то макрос `BOOST_COMP_GNUC` определяется как `0`.

Благодаря такому подходу вам не нужно проверять, определен ли сам макрос:

```
#if defined(BOOST_COMP_GNUC) // Неправильно!

// GCC specific

#endif
```

Макросы библиотеки `Boost.Predef` всегда определены, что избавляет вас от необходимости набирать `define()` или `def` в `#ifdef`.

Дополнительно...

Библиотека `Boost.Predef` также содержит макросы для определения ОС, архитектуры, реализации стандартной библиотеки и некоторых аппаратных возможностей. Подход с использованием макросов, которые всегда определены, позволяет вам писать сложные выражения намного короче:

```
#include <boost/predef/os.h>
#include <boost/predef/compiler.h>

#if BOOST_COMP_GNUC && BOOST_OS_LINUX && !BOOST_OS_ANDROID

// Делаем что-то для non-Android Linux.

#endif
```

Теперь лучшая часть. Библиотеку `Boost.Predef` можно использовать на компиляторах C, C++ и Objective-C. Если хотите, используйте ее в своих проектах, написанных не на языке C++. Стандарт C++ лишен функциональности библиотеки `Boost.Predef`.

См. также

- Прочитайте официальную документацию по `Boost.Predef` для получения дополнительной информации о ее возможностях: http://boost.org/libs/predef;
- следующий рецепт познакомит вас с библиотекой `Boost.Config`, которая намного старше, немного менее красива, но гораздо более функциональна.

Обнаружение поддержки 128-битных целых чисел

Некоторые компиляторы поддерживают расширенные арифметические типы, такие как 128-разрядные числа с плавающей точкой или целые числа. Давайте кратко рассмотрим, как их использовать с помощью Boost.

Мы будем создавать метод, который принимает три параметра и возвращает умноженное значение этих методов. Если компилятор поддерживает 128-битные целые числа, то мы их используем. Если компилятор поддерживает тип `long long`, тогда мы используем его; в противном случае нам нужно выдать ошибку времени компиляции.

Подготовка

Требуются только базовые знания C++.

Как это делается...

Что нам нужно для работы со 128-битными целыми числами? Макросы, которые показывают, что они доступны, и несколько `typedef` для создания переносимых имен типов на разных платформах.

1. Подключите заголовочный файл:

```
#include <boost/config.hpp>
```

2. Теперь нам нужно определить поддержку 128-битных целых чисел:

```
#ifdef BOOST_HAS_INT128
```

3. Добавьте typedef и реализуйте метод следующим образом:

```
typedef boost::int128_type int_t;
typedef boost::uint128_type uint_t;

inline int_t mul(int_t v1, int_t v2, int_t v3) {
    return v1 * v2 * v3;
}
```

4. Для компиляторов, которые не поддерживают 128-битные целые числа и тип long long, можно выдать ошибку времени компиляции:

```
#else // #ifdef BOOST_HAS_INT128

#ifdef BOOST_NO_LONG_LONG
#error "This code requires at least int64_t support"
#endif
```

5. Теперь нам нужно предоставить реализацию для компиляторов без поддержки 128-битных целых чисел с использованием int64:

```
struct int_t { boost::long_long_type hi, lo; };
struct uint_t { boost::ulong_long_type hi, lo; };

inline int_t mul(int_t v1, int_t v2, int_t v3) {
    // Математические вычисления, написанные вручную.
    // ...
}

#endif // #ifdef BOOST_HAS_INT128
```

Как это работает...

Заголовочный файл <boost/config.hpp> содержит множество макросов для описания возможностей компилятора и платформы. В этом примере мы использовали BOOST_HAS_INT128 для обнаружения поддержки 128-битных целых чисел и BOOST_NO_LONG_LONG для обнаружения поддержки 64-битных целых чисел.

Как видно из примера, в Boost есть typedef для 64-битных целых чисел со знаком и без:

```
boost::long_long_type
boost::ulong_long_type
```

А также typedef для 128-битных целых чисел со знаком и без:

```
boost::int128_type
boost::uint128_type
```

Дополнительно...

C++11 поддерживает 64-битные типы через встроенные типы long long int и unsigned long long int. К сожалению, не все компиляторы поддерживают C++11, поэтому макрос BOOST_NO_LONG_LONG может быть вам полезен.

128-битные целые числа не являются частью стандарта C++, поэтому `typedef` и макросы из Boost являются одним из способов написания переносимого кода.

В комитете по стандартизации C++ продолжается работа по добавлению целых чисел заданной во время компиляции ширины. Когда эта работа будет завершена, вы сможете создавать 128-битные, 512-битные и даже 8 388 608-битные целые числа.

См. также

- Прочтите рецепт «*Обнаружение и обход отключенной динамической идентификации типа данных*» для получения дополнительной информации о `Boost.Config`;
- прочтите официальную документацию по `Boost.Config` на странице http://boost.org/libs/config для получения дополнительной информации о ее возможностях;
- в Boost есть библиотека, которая позволяет создавать типы с неограниченной точностью. Перейдите по ссылке http://boost.org/libs/multiprecision.

Обнаружение и обход отключенной динамической идентификации типа данных

Некоторые компании и проекты предъявляют особые требования к своему коду C++, такие как успешная компиляция без динамической идентификации типа данных (RTTI).

В этом небольшом рецепте мы не только обнаружим отключенную динамическую идентификацию, но и напишем Boost-подобную библиотеку, которая хранит информацию о типах и сравнивает типы во время выполнения, даже без `typeid`.

Подготовка

Для этого рецепта необходимы базовые знания об использовании динамической идентификации типа данных в C++.

Как это делается...

Обнаружение отключенной динамической идентификации типа данных, хранение информации о типах и сравнение типов во время выполнения – приемы, которые широко используются в библиотеках Boost.

1. Для этого сначала нужно подключить следующий заголовочный файл:

```
#include <boost/config.hpp>
```

2. Давайте сначала посмотрим на ситуацию, когда динамическая идентификация типа данных включена и доступен класс `std::type_index`:

```
#if !defined(BOOST_NO_RTTI) \
    && !defined(BOOST_NO_CXX11_HDR_TYPEINDEX)

#include <typeindex>
using std::type_index;
```

```
template <class T>
type_index type_id() {
    return typeid(T);
}
```

3. В противном случае нам нужно создать собственный класс `type_index`:

```
#else

#include <cstring>
#include <iosfwd> // std::basic_ostream
#include <boost/current_function.hpp>

struct type_index {
    const char * name_;

    explicit type_index(const char* name)
        : name_(name)
    {}

    const char* name() const { return name_; }
};

inline bool operator == (type_index v1, type_index v2) {
    return !std::strcmp(v1.name_, v2.name_);
}

inline bool operator != (type_index v1, type_index v2) {
    return !(v1 == v2);
}
```

4. Последний шаг – определить функцию `type_id`:

```
template <class T>
inline type_index type_id() {
    return type_index(BOOST_CURRENT_FUNCTION);
}

#endif
```

5. Теперь мы можем сравнивать типы:

```
#include <cassert>

int main() {
    assert(type_id<unsigned int>() == type_id<unsigned>());
    assert(type_id<double>() != type_id<long double>());
}
```

Как это работает...

Макрос `BOOST_NO_RTTI` определяется, если динамическая идентификация типа данных отключена, а макрос `BOOST_NO_CXX11_HDR_TYPEINDEX` определяется, когда компилятор не имеет заголовка `<typeindex>` и класса `std::type_index`.

Рукописная структура `type_index` из *этапа 3* предыдущего раздела содержит только указатель на некую строку; на самом деле здесь нет ничего интересного.

Посмотрите на макрос BOOST_CURRENT_FUNCTION. Он возвращает полное имя текущей функции, включая параметры шаблона, аргументы и тип возвращаемого значения.

Например, при вызове type_id<double>() макрос вернет:

```
type_index type_id() [with T = double]
```

Таким образом, для любого другого типа BOOST_CURRENT_FUNCTION возвращает иную строку. Поэтому переменные type_index для разных типов в примере не равны.

Поздравляю! Мы только что заново переизобрели бóльшую часть функционала библиотеки Boost.TypeIndex. Удалите весь код из *этапов* с 1 по 4 и слегка измените код в *этапе 5* для использования библиотеки Boost.TypeIndex:

```
#include <boost/type_index.hpp>

void test() {
    using boost::typeindex::type_id;

    assert(type_id<unsigned int>() == type_id<unsigned>());
    assert(type_id<double>() != type_id<long double>());
}
```

Дополнительно...

Конечно, это не все, что умеет библиотека Boost.TypeIndex. Она позволяет получать удобочитаемое имя типа независимо от платформы, обходить платформоспецифичные проблемы, дает возможность применять собственную реализацию динамической идентификации типа данных, использовать динамическую идентификацию типов на этапе компиляции и другие вещи.

Разные компиляторы имеют разные макросы для получения полного имени функции. Использование макросов от Boost является наиболее переносимым решением. Макрос BOOST_CURRENT_FUNCTION возвращает имя во время компиляции, поэтому он подразумевает минимальные потери времени выполнения.

В C++11 есть магический идентификатор __func__, который вычисляется по имени текущей функции. Однако результатом __func__ является только имя функции, в то время как BOOST_CURRENT_FUNCTION изо всех сил старается также показать параметры функции, включая шаблоны.

См. также

- Прочитайте последующие рецепты для получения дополнительной информации по Boost.Config;
- перейдите на страницу http://github.com/boostorg/type_index, чтобы просмотреть исходные коды библиотеки Boost.TypeIndex;
- прочтите официальную документацию по Boost.Config на странице http://boost.org/libs/config;
- прочитайте официальную документацию по библиотеке Boost.TypeIndex по адресу http://boost.org/libs/type_index;
- рецепт «*Получение удобочитаемого имени типа*» главы 1 «*Приступаем к написанию приложения*» познакомит вас с другими возможностями Boost.TypeIndex.

Написание метафункций с использованием более простых методов

Глава 4 «*Уловки времени компиляции*» и глава 8 были посвящены метапрограммированию. Если вы пытались использовать приемы из этих глав, то могли заметить, что написание метафункции может занять много времени. Таким образом, перед написанием переносимой реализации, возможно, будет полезно поэкспериментировать с метафункциями, используя более удобные методы, такие как `constexpr` из C++11.

В этом рецепте мы рассмотрим, как обнаружить поддержку `constexpr`.

Подготовка

Функции `constexpr`– это функции, которые можно вычислять во время компиляции. Это все, что нам нужно знать для данного рецепта.

Как это делается...

Давайте посмотрим, как можно определить поддержку компилятором функций `constexpr`.

1. Как и в других рецептах из этой главы, мы начинаем со следующего заголовочного файла:

```
#include <boost/config.hpp>
```

2. Напишите функцию `constexpr`:

```
#if !defined(BOOST_NO_CXX11_CONSTEXPR) \
    && !defined(BOOST_NO_CXX11_HDR_ARRAY)

template <class T>
constexpr int get_size(const T& val) {
    return val.size() * sizeof(typename T::value_type);
}
```

3. Выводим ошибку, если возможности C++11 отсутствуют:

```
#else
#error "This code requires C++11 constexpr and std::array"
#endif
```

4. Вот и все. Теперь можно писать такой код:

```
#include <array>

int main() {
    std::array<short, 5> arr;
    static_assert(get_size(arr) == 5 * sizeof(short), "");

    unsigned char data[get_size(arr)];
}
```

Как это работает...

Макрос BOOST_NO_CXX11_CONSTEXPR определяется, когда доступны функции constexpr.

Ключевое слово constexpr сообщает компилятору, что функцию можно вычислить во время компиляции, если все входные данные для этой функции являются константами времени компиляции. C++11 накладывает множество ограничений на то, что может делать функция constexpr. C++14 и C++20 убрали многие из этих ограничений.

Макрос BOOST_NO_CXX11_HDR_ARRAY определяется, когда доступны класс C++11 std::array и заголовок <array>.

Дополнительно...

Однако есть и другие полезные и интересные макросы для constexpr, а именно:

- макрос BOOST_CONSTEXPR раскрывается в constexpr или в ничто, если constexpr не поддерживается компилятором;
- макрос BOOST_CONSTEXPR_OR_CONST раскрывается в constexpr или const;
- макрос BOOST_STATIC_CONSTEXPR эквивалентен static BOOST_CONSTEXPR_OR_CONST.

С помощью этих макросов можно написать код, который использует возможности константных выражений C++11, если они доступны:

```
template <class T, T Value>
struct integral_constant {
    BOOST_STATIC_CONSTEXPR T value = Value;
    BOOST_CONSTEXPR operator T() const {
        return this->value;
    }
};
```

Теперь мы можем использовать integral_constant, как показано ниже:

```
char array[integral_constant<int, 10>()];
```

В этом примере для получения размера массива вызывается BOOST_CONSTEXPR operator T().

Константные выражения C++11 могут улучшить скорость компиляции и диагностическую информацию в случае ошибки. Это хорошая возможность. Если вашей функции требуется менее ограниченный constexpr из C++14, можете использовать макрос BOOST_CXX14_CONSTEXPR. Он раскрывается в constexpr, только если constexpr из C++14 поддерживается компилятором, а в противном случае раскрывается в ничто.

См. также

- Более подробную информацию об использовании constexpr можно прочитать по адресу http://en.cppreference.com/w/cpp/language/constexpr;
- прочтите официальную документацию по Boost.Config для получения дополнительной информации о макросах на странице http://boost.org/libs/config.

Уменьшение размера кода и повышение производительности пользовательских типов в C++11

Начиная с C++11 есть очень специфическая логика при использовании пользовательских типов в контейнерах стандартной библиотеки. Некоторые контейнеры используют присваивание перемещением и конструктор перемещения только в том случае, если конструктор перемещения пользовательского типа не выбрасывает исключения или отсутствует конструктор копирования.

Давайте посмотрим, как можно переносимо гарантировать компилятору, что у класса `move_nothrow` есть оператор присваивания перемещения, который не выбрасывает исключения, и конструктор перемещения, который также не выбрасывает исключения.

Подготовка

Для этого рецепта требуются базовые знания rvalue-ссылок. Знание контейнеров стандартной библиотеки также вам пригодится.

Как это делается...

Давайте посмотрим, как можно улучшить наши классы C++, используя Boost.

1. Все, что нам нужно сделать, – это пометить оператор присваивания `move_nothrow` и конструктор `move_nothrow` с помощью макроса `BOOST_NOEXCEPT`:

```
#include <boost/config.hpp>

class move_nothrow {
    // Здесь идут члены класса.
    // ...

public:
    move_nothrow() BOOST_NOEXCEPT;
    move_nothrow(move_nothrow&&) BOOST_NOEXCEPT
        // Инициализация членов идет здесь.
        // ...
    {}

    move_nothrow& operator=(move_nothrow&&) BOOST_NOEXCEPT {
        // Здесь идет реализация
        // ...
        return *this;
    }

    move_nothrow(const move_nothrow&);
    move_nothrow& operator=(const move_nothrow&);
};
```

2. Можем использовать наш класс с `std::vector`:

```
#include <vector>

int main() {
    std::vector<move_nothrow> v(10);
```

```
        v.push_back(move_nothrow());
    }
```

3. Можно убедиться, что контейнером используется именно конструктор перемещения. Если мы удалим макрос BOOST_NOEXCEPT из конструктора перемещения, то получим ошибку, потому что начинает использоваться конструктор копирования, а мы не предоставили определения для него:

```
undefined reference to `move_nothrow::move_nothrow(move_nothrow
const&)
```

Как это работает...

Макрос BOOST_NOEXCEPT раскрывается в noexcept на компиляторах, которые его поддерживают. Контейнеры стандартной библиотеки используют свойства типа из заголовочного файла <type_traits>, чтобы определить, выбрасывает конструктор исключение или нет. Свойства типа принимают решение главным образом на основе спецификаторов noexcept.

Почему без BOOST_NOEXCEPT мы получаем ошибку? Свойство типа возвращает, что конструктор перемещения move_nothrow может кинуть исключение, поэтому std::vector пытается использовать конструктор копирования move_nothrow, который не определен.

Дополнительно...

Макрос BOOST_NOEXCEPT также уменьшает размер двоичного файла, если определение функции находится в отдельном исходном файле.

```
// В заголовочном файле.
int foo() BOOST_NOEXCEPT;

// В исходном файле.
int foo() BOOST_NOEXCEPT {
    return 0;
}
```

С BOOST_NOEXCEPT компилятор знает, что функция не выбрасывает исключения, и поэтому нет необходимости генерировать код, который их обрабатывает.

Если функция, помеченная noexcept, выбрасывает исключение, ваша программа аварийно завершит работу без вызова деструкторов для созданных объектов.

См. также

- Документ, объясняющий, почему конструкторам перемещения разрешено выбрасывать исключения и как контейнеры должны перемещать объекты, доступен по адресу http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2010/n3050.html;
- прочтите официальную документацию по Boost.Config, где приводятся дополнительные примеры макросов наподобие BOOST_NOEXCEPT: http://boost.org/libs/config.

Переносимый способ экспорта и импорта функций и классов

Почти все современные языки имеют возможность создавать библиотеки – коллекции классов и методов, с четко определенным интерфейсом. C++ не является исключением из этого правила. В нем есть два типа библиотек: статические и библиотеки времени выполнения,которые также называют динамическими (англ. shared library, dynamic library). Но написание библиотек в C++ – задача непростая. Разные платформы имеют различные методы для описания того, какие символы нужно экспортировать из динамической библиотеки.

Давайте посмотрим, как управлять видимостью символов переносимым способом с помощью Boost.

Подготовка

В этом рецепте может быть полезен опыт создания динамических и статических библиотек.

Как это делается...

Код для этого рецепта состоит из двух частей. Первая часть – это сама библиотека. Вторая часть – код, который использует эту библиотеку. Обе части применяют один и тот же заголовочный файл, в котором объявлены методы библиотеки. Управление видимостью символов переносимым способом с помощью Boost является простым и может быть выполнено с помощью следующих шагов.

1. Нам нужны определения из этого заголовочного файла:

```
#include <boost/config.hpp>
```

2. Приведенный ниже код также нужно добавить в заголовочный файл:

```
#if defined(MY_LIBRARY_LINK_DYNAMIC)
#   if defined(MY_LIBRARY_COMPILATION)
#       define MY_LIBRARY_API BOOST_SYMBOL_EXPORT
#   else
#       define MY_LIBRARY_API BOOST_SYMBOL_IMPORT
#   endif
#else
#   define MY_LIBRARY_API
#endif
```

3. Теперь все объявления функций динамической библиотеки должны использовать макрос MY_LIBRARY_API:

```
int MY_LIBRARY_API foo();

class MY_LIBRARY_API bar {
public:
    /* ... */
    int meow() const;
};
```

4. Исключения должны объявляться с помощью `BOOST_SYMBOL_VISIBLE`; в противном случае в коде, который использует библиотеку, их зачастую невозможно будет перехватить по имени или имени базового класса:

```
#include <stdexcept>

struct BOOST_SYMBOL_VISIBLE bar_exception
    : public std::exception
{};
```

5. Исходные файлы библиотеки должны включать в себя заголовочный файл:

```
#define MY_LIBRARY_COMPILATION
#include "my_library.hpp"
```

6. Определения методов также должны находиться в исходных файлах библиотеки:

```
int MY_LIBRARY_API foo() {
   // Здесь идет реализация.
   // ...
   return 0;
}

int bar::meow() const {
    throw bar_exception();
}
```

7. Теперь можно использовать библиотеку, как показано ниже:

```
#include "../06_A_my_library/my_library.hpp"
#include <cassert>

int main() {
    assert(foo() == 0);
    bar b;
    try {
        b.meow();
        assert(false);
    } catch (const bar_exception&) {}
}
```

Как это работает...

Вся работа выполняется на *этапе 2*. Там мы определяем макрос `MY_LIBRARY_API`, применяемый к классам и методам, которые хотим экспортировать из нашей статической или динамической библиотеки. На *этапе 2* мы проверяем, определен ли макрос `MY_LIBRARY_LINK_DYNAMIC`. Если он не определен, то создаем статическую библиотеку, и определять `MY_LIBRARY_API` нет необходимости.

Разработчик должен позаботиться о макросе `MY_LIBRARY_LINK_DYNAMIC`! Он не выставит себя сам. Если мы создаем динамическую библиотеку, нам нужно заставить систему сборки определить его.

Если MY_LIBRARY_LINK_DYNAMIC определен, мы создаем динамическую библиотеку, и именно здесь начинаются сложности. Вы как разработчик при сборке библиотеки должны сообщить компилятору, что сейчас мы экспортируем функцию для пользователя. Пользователь при сборке должен сообщить компилятору, какие методы из библиотеки импортируются. Чтобы у нас был один заголовочный файл для импорта и экспорта библиотеки, мы используем этот код:

```
#if defined(MY_LIBRARY_COMPILATION)
#    define MY_LIBRARY_API BOOST_SYMBOL_EXPORT
#else
#    define MY_LIBRARY_API BOOST_SYMBOL_IMPORT
#endif
```

При экспорте библиотеки (или, другими словами, ее компиляции) мы должны определить макрос MY_LIBRARY_COMPILATION. Это приведет к тому, что MY_LIBRARY_API раскроется в BOOST_SYMBOL_EXPORT. *См. этап 5*, где мы определили MY_LIBRARY_COMPILATION перед подключением заголовка my_library.hpp.

Если MY_LIBRARY_COMPILATION не определен, заголовок подключается пользователем, который ничего не знает об этом макросе. А если заголовок подключен пользователем, символы должны импортироваться из библиотеки и MY_LIBRARY_API раскрывается в BOOST_SYMBOL_IMPORT.

Макрос BOOST_SYMBOL_VISIBLE должен использоваться только для тех классов, которые не экспортируются, а используются динамической идентификацией типа данных. Примерами таких классов являются исключения и классы, приводимые с использованием dynamic_cast.

Дополнительно...

Некоторые компиляторы экспортируют все символы по умолчанию, но предоставляют флаги для отключения такого поведения. Например, GCC и Clang в Linux предоставляют флаг -fvisibility=hidden. Настоятельно рекомендуется использовать данные флаги, потому что это приводит к меньшему размеру двоичного файла, более быстрой загрузке динамических библиотек и лучшему логическому структурированию двоичного файла. Некоторые межпроцедурные оптимизации могут работать лучше, когда экспортируется меньше символов. В стандарте C++ нет стандартного способа описания видимости. Надеемся, что когда-нибудь они появятся, но до тех пор мы должны использовать макросы из Boost.

См. также

- Прочтите эту главу с самого начала, чтобы увидеть дополнительные примеры использования Boost.Config;
- ознакомьтесь с официальной документацией по Boost.Config для получения полного списка макросов и их описания по адресу http://boost.org/libs/config.

Обнаружение версии Boost и получение новейших функций

Boost активно развивается, поэтому каждый выпуск содержит новые функции и библиотеки. Некоторые люди хотят иметь библиотеки, которые компилируются для разных версий Boost, а также хотят использовать некоторые функции новых версий.

Давайте посмотрим на журнал изменений `boost::lexical_cast`. Согласно этому журналу, в Boost версии 1.53 есть функция `lexical_cast(const CharTyp* chars, std ::size_t count)`. Наша задача для этого рецепта будет заключаться в том, чтобы использовать эту функцию для новых версий Boost и обходиться без нее для более старых версий.

Подготовка

Требуются только базовые знания C++ и библиотеки `Boost.LexicalCast`.

Как это делается...

Все, что нам нужно сделать, – это получить информацию о версии Boost и использовать ее для написания оптимального кода. Это можно сделать, как показано далее.

1. Нам нужно подключить заголовочные файлы, содержащие версию Boost и `boost::lexical_cast`:

```
#include <boost/version.hpp>
#include <boost/lexical_cast.hpp>
```

2. Мы используем новую функцию `Boost.LexicalCast`, если она доступна:

```
#if (BOOST_VERSION >= 105200)

int to_int(const char* str, std::size_t length) {
    return boost::lexical_cast<int>(str, length);
}
```

3. В противном случае мы должны сначала скопировать данные в `std::string`:

```
#else

int to_int(const char* str, std::size_t length) {
    return boost::lexical_cast<int>(
        std::string(str, length)
    );
}
#endif
```

4. Теперь мы можем использовать код, как показано здесь:

```
#include <cassert>

int main() {
```

```
    assert(to_int("10000000", 3) == 100);
}
```

Как это работает...

Макрос BOOST_VERSION содержит версию Boost, написанную в следующем формате: одна цифра для старшей версии, затем три цифры для младшей версии, а потом две цифры для уровня исправлений. Например, Boost версии 1.73.1 будет содержать номер 107301 в макросе BOOST_VERSION.

Итак, на *этапе 2* мы проверяем версию Boost и выбираем правильную реализацию функции to_int в соответствии со способностями Boost.LexicalCast.

Дополнительно...

Наличие макроса версии является обычной практикой для больших библиотек. Некоторые из библиотек Boost позволяют вам указать версию библиотеки, например Boost.Thread и ее макрос BOOST_THREAD_VERSION.

Кстати, в C++ тоже есть макрос версии. Значение макроса __cplusplus позволяет отличить стандарт, предшествующий C++11, от C++11, C++11 от C++14, C++17 или C++20. В настоящее время он может быть выставлен в одно из следующих значений: 199711L, 201103L, 201402L, 201703L или 202002L. Значение макроса означает год и месяц утверждения стандарта C++.

См. также

- Прочтите рецепт «*Создание потока выполнения*» главы 5 «*Многопоточность*» для получения дополнительной информации о BOOST_THREAD_VERSION и о том, как он влияет на библиотеку Boost.Thread или документацию по адресу http://boost.org/libs/thread;
- прочтите эту главу с самого начала или подумайте о том, чтобы ознакомиться с официальной документацией по Boost.Config по адресу http://boost.org/libs/config.

Глава 11

Работа с системой

Темы, которые мы рассмотрим в этой главе:

- перечисление файлов в каталоге;
- стирание и создание файлов и каталогов;
- написание и использование плагинов;
- получение backtrace – текущей последовательности вызовов;
- быстрая передача данных из одного процесса в другой;
- синхронизация межпроцессного взаимодействия;
- использование указателей в общей памяти;
- самый быстрый способ чтения файлов;
- сопрограммы – сохранение состояния и откладывание выполнения.

Вступление

Различные ОС могут предоставлять похожий функционал через системные вызовы, различающиеся как именами, так и аргументами. Boost предоставляет переносимые и безопасные обертки для этих вызовов.

Эта глава посвящена работе с операционной системой. Мы уже видели, как работать с передачей данных по сети и сигналами, в главе 6 «*Управление задачами*». В этой главе мы подробнее рассмотрим файловую систему, создание и удаление файлов. Мы увидим, как можно передавать данные между различными системными процессами, как читать файлы на максимальной скорости и выполнять другие трюки.

Перечисление файлов в каталоге

Существуют функции и классы стандартной библиотеки для чтения и записи данных в файлы. Но до появления C++17 в ней не было функций для вывода списка файлов в каталоге, получения типа файла или получения прав доступа к файлу.

Давайте посмотрим, как можно исправить эту несправедливость с помощью Boost. Мы будем создавать программу, которая перечисляет имена файлов, права на запись и типы файлов в текущем каталоге.

Подготовка

Знание основ C++ более чем достаточно для использования этого рецепта. Этот рецепт требует линковки с библиотеками `boost_system` и `boost_filesystem`.

Как это делается...

Этот и последующий рецепты посвящены переносимым оберткам для работы с файловой системой.

1. Нам нужно подключить следующие два заголовочных файла:

```
#include <boost/filesystem/operations.hpp>
#include <iostream>
```

2. Теперь нужно указать каталог:

```
int main() {
    boost::filesystem::directory_iterator begin("./");
```

3. После указания каталога переберите его содержимое:

```
    boost::filesystem::directory_iterator end;
    for (; begin != end; ++ begin) {
```

4. Следующий шаг – получение информации о файле:

```
        boost::filesystem::file_status fs =
            boost::filesystem::status(*begin);
```

5. Теперь выведите информацию о файле:

```
        switch (fs.type()) {
        case boost::filesystem::regular_file:
            std::cout << "FILE ";
            break;
        case boost::filesystem::symlink_file:
            std::cout << "SYMLINK ";
            break;
        case boost::filesystem::directory_file:
            std::cout << "DIRECTORY ";
            break;
        default:
            std::cout << "OTHER ";
            break;
        }
        if (fs.permissions() & boost::filesystem::owner_write) {
            std::cout << "W ";
        } else {
            std::cout << " ";
        }
```

6. Последний шаг – вывод имени файла:

```
        std::cout << *begin << '\n';
    } /*for*/
} /*main*/
```

Готово. Теперь, если мы запустим программу, она выведет что-то вроде этого:

```
FILE W "./main.o"
FILE W "./listing_files"
DIRECTORY W "./some_directory"
FILE W "./Makefile"
```

Как это работает...

Функции и классы Boost.Filesystem просто оборачивают системные вызовы для работы с файлами.

Обратите внимание на использование знака /"." *на этапе 2*. Системы POSIX используют косую черту для указания путей; Windows по умолчанию использует обратную косую черту. Тем не менее Windows также понимает косую черту, а даже если бы не понимала, то библиотека Boost позаботилась бы о неявном преобразовании формата пути.

Посмотрите на *этап 3*, где мы вызываем конструктор по умолчанию для класса boost::filesystem::directory_iterator. Этот конструктор работает по аналогии с конструктором по умолчанию класса std::istream_iterator, – создает итератор конца диапазона.

Этап 4 сложен не потому, что эту функцию трудно понять, а из-за того, что происходит много преобразований. Разыменование итератора begin возвращает boost::filesystem::directory_entry, который неявно преобразуется в boost::filesystem::path, использующийся в качестве параметра для функции boost::filesystem::status. На самом деле можно написать намного лучше:

```
boost::filesystem::file_status fs = begin->status();
```

Внимательно прочитайте справочную документацию, чтобы избежать ненужных неявных преобразований.

Этап 5 очевиден, поэтому мы переходим к *этапу 6*, где неявное преобразование в boost::filesystem::path происходит снова. Более явное решение выглядит так:

```
std::cout << begin->path() << '\n';
```

Здесь begin->path() возвращает константную ссылку на переменную boost::filesystem::path, которая содержится в boost::filesystem::directory_entry.

Дополнительно...

Boost.Filesystem является частью C++17. Все содержимое в C++17 находится в одном заголовочном файле <filesystem> в пространстве имен std::filesystem::. Версия <filesystem> стандартной библиотеки несколько отличается от Boost-версии, в основном за счет использования перечислений с областью видимости (enum class) там, где Boost.Filesystem использовала просто перечисление без области видимости.

Есть класс directory_entry, который обеспечивает кеширование информации о файловой системе. Так что если вы много работаете с файловой системой и запрашиваете различную информацию, попробуйте использовать directory_entry для лучшей производительности.

Как и в случае с другими библиотеками Boost, `Boost.Filesystem` работает с компиляторами для стандарта, предшествующего C++17, и даже с компиляторами для стандарта, предшествующего C++11.

См. также

- Рецепт «*Стирание и создание файлов и каталогов*» покажет еще один пример использования `Boost.Filesystem`;
- прочтите официальную документацию по `Boost.Filesystem`, чтобы получить больше информации о ее возможностях на странице http://boost.org/libs/filesystem;
- рабочую версию проекта C++17 можно найти по адресу http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/n4659.pdf.

Стирание и создание файлов и каталогов

Давайте рассмотрим следующие строки кода:

```
std::ofstream ofs("dir/subdir/file.txt");
ofs << "Boost.Filesystem is fun!";
```

В этих строках мы пытаемся записать что-то в файл `file.txt` в каталоге `dir/subdir`. Если такой директории нет, эта попытка будет неудачной.

В этом рецепте мы создадим каталог и подкаталог, запишем некие данные в файл и попробуем создать символическую ссылку. Если создание символической ссылки не удается, то мы удалим созданные сущности. В примере мы также будем избегать использования исключений в качестве механизма сообщения об ошибках, отдавая предпочтение чему-то вроде кодов возврата.

Давайте посмотрим, как можно сделать это элегантно, используя Boost.

Подготовка

Для этого рецепта требуются базовые знания C++ и класса `std::ofstream`.

`Boost.Filesystem` не является библиотекой header-only, поэтому код в этом рецепте требуется линковать с библиотеками `boost_system` и `boost_filesystem`.

Как это делается...

Мы продолжаем работать с переносимыми обертками для файловой системы и в этом рецепте посмотрим, как изменить содержимое каталога.

1. Как всегда, нам нужно подключить несколько заголовочных файлов:

```
#include <boost/filesystem/operations.hpp>
#include <cassert>
#include <fstream>
```

2. Теперь нам нужна переменная для хранения ошибок (если таковые появятся):

```
int main() {
    boost::system::error_code error;
```

3. Мы также создадим каталоги:

```
    boost::filesystem::create_directories("dir/subdir", error);
    assert(!error);
```

4. Затем запишем данные в файл:

```
    std::ofstream ofs("dir/subdir/file.txt");
    ofs << "Boost.Filesystem is fun!";
    assert(ofs);
    ofs.close();
```

5. Попытаемся создать символическую ссылку:

```
    boost::filesystem::create_symlink(
        "dir/subdir/file.txt", "symlink", error);
```

6. Потом нам нужно проверить, что файл доступен через эту ссылку:

```
    if (!error) {
        std::cerr << "Symlink created\n";
        assert(boost::filesystem::exists("symlink"));
```

7. Удалим файл, если создание символической ссылки завершится неудачно:

```
    } else {
        std::cerr << "Failed to create a symlink\n";

        boost::filesystem::remove_all("dir", error);
        assert(!error);
        boost::filesystem::remove("symlink", error);
        assert(!error);
    } /*if (!error)*/
} /*main*/
```

Как это работает...

Мы видели, как работает `boost::system::error_code`, практически во всех рецептах главы 6 «*Управление задачами*». Он может хранить информацию об ошибках и широко используется во всех библиотеках Boost.

Если вы не предоставите экземпляр `boost::system::error_code` для функций `Boost.Filesystem`, код будет компилироваться. В этом случае при возникновении ошибки будет выброшено исключение `boost::filesystem::filesystem_error`.

Внимательно посмотрите на *этап 3*. Мы использовали функцию `boost::filesystem::create_directories` вместо `boost::filesystem::create_directory`, потому что последняя не может создавать вложенные подкаталоги. Та же самая история с `boost::filesystem::remove_all` и `boost::filesystem::remove`. Первая удаляет каталоги, которые могут содержать файлы и подкаталоги. Вторая удаляет один файл.

Остальные шаги просты для понимания и не должны вызывать проблем.

Дополнительно...

Класс `boost::system::error_code` является частью C++11. Его можно найти в заголовочном файле `<system_error>` в пространстве имен `std::`. Классы `Boost.Filesystem` являются частью C++17.

Наконец, небольшая рекомендация для тех, кто собирается использовать `Boost.Filesystem`. Когда ошибки при работе с файловой системой являются частым явлением, или приложение требует высокой отзывчивости/производительности, используйте класс `boost::system::error_codes`. В противном случае для обработки ошибок перехват исключений будет предпочтительнее и надежнее.

См. также

Рецепт «*Перечисление файлов в каталоге*» также содержит информацию о `Boost.Filesystem`. Прочтите официальную документацию по адресу http://boost.org/libs/filesystem, где приводится больше информации и примеров.

Написание и использование плагинов

Вот вам задачка: мы хотим позволить пользователям писать код для расширения функционала нашей программы, но не хотим давать им исходники. Другими словами, мы хотим сказать пользователям: «*Напишите функцию X и упакуйте ее в общую библиотеку. Мы сможем использовать вашу функцию вместе с функциями других пользователей!*»

Вы сталкиваетесь с этим приемом в повседневной жизни: ваш браузер использует его для подключения плагинов, ваш текстовый редактор – для подсветки синтаксиса, игры используют **загрузку динамической библиотеки для загружаемого контента** (англ. **DLC**) и для добавления контента игрока, веб-серверы используют плагины для поддержки шифрования или аутентификации и т. д.

Каковы требования для функции пользователя и как можно использовать эту функцию, не линкуясь с его библиотекой?

Подготовка

Для этого рецепта требуются базовые знания C++. Также необходимо прочитать рецепт «*Переносимый способ экспорта и импорта функций и классов*» главы 10.

Как это делается...

Прежде всего вы должны описать «контракт» по написанию плагина.

1. Документируйте требования к интерфейсу плагина. Например, вы можете сказать, что все плагины должны экспортировать функцию с именем `greet` и что эта функция должна принимать `const std::string&` и возвращать `std::string`.
2. После этого пользователи могут начать писать плагины или общую библиотеку следующим образом:

```
#include <string>
#include <boost/config.hpp>

#define API extern "C" BOOST_SYMBOL_EXPORT

API std::string greeter(const std::string& name) {
    return "Good to meet you, " + name + ".";
}
```

3. Ваш программный код для загрузки динамической библиотеки должен содержать заголовочный файл из `Boost.DLL`:

```
#include <boost/dll/shared_library.hpp>
```

4. Код для загрузки библиотеки должен выглядеть так:

```
int main() {
    boost::filesystem::path plugin_path = /* path-to-pligin */;

    boost::dll::shared_library plugin(
        plugin_path,
        boost::dll::load_mode::append_decorations
    );
```

5. Получение функции пользователя должно выглядеть так:

```
    auto greeter = plugin.get<std::string(const std::string&)>("greeter");
```

6. Готово. Теперь можно использовать эту функцию:

```
    std::cout << greeter("Sally Sparrow");
}
```

В зависимости от загруженного плагина у вас будут разные результаты:

```
plugin_hello:

       Good to meet you, Sally Sparrow.

plugin_do_not:

      They are fast. Faster than you can believe. Don't turn
      your back, don't look away, and don't blink. Good luck, Sally Sparrow.
```

Как это работает...

На *этапе 2* есть небольшая хитрость. Когда вы объявляете функцию как `extern "C"`, это означает, что компилятор не должен **искажать** (англ. **mangle**) имя функции. Другими словами, на *этапе 2* мы просто создаем функцию, которая имеет имя `greet` и которая экспортируется с этим именем из динамической библиотеки.

На этапе 4 мы создаем переменную `boost::dll::shared_library` с именем `plugin`. Конструктор этой переменной загружает общую библиотеку по указанному пути в адресное пространство текущего исполняемого файла. На этапе 5 мы ищем функцию с именем `greet` в плагине, а также указываем, что функция имеет сигнатуру `std::string(const std::string&)`, и сохраняем указатель на эту функцию в переменной `greet`.

Готово! Отныне мы можем использовать переменную `greet` в качестве функции, пока переменная `plugin` и все ее копии не уничтожены.

Вы можете экспортировать несколько функций из динамической библиотеки и даже экспортировать переменные.

Будьте осторожны! Всегда динамически линкуйте библиотеки С и C++ с плагином и вашим основным исполняемым файлом, потому что в противном случае ваше приложение даст сбой. Всегда используйте одинаковые или ABI-совместимые (бинарно-совместимые) версии библиотек С и C++ в плагинах и в приложении. В противном случае приложение даст сбой. Прочитайте документацию, где приводятся примеры типичного неправильного использования!

Дополнительно...

`Boost.DLL` – сравнительно новая библиотека; она появилась в Boost версии 1.61. Что мне больше всего нравится в ней – это возможность добавлять специфичные для платформы декорации к имени динамической библиотеки. Например, этот код, в зависимости от платформы, попытается загрузить `"./some/path/libplugin_name.so"`, `"./some/path/plugin_name.dll"` или `"./some/path/libplugin_name.dll"`:

```
boost::dll::shared_library lib(
    "./some/path/plugin_name",
    boost::dll::load_mode::append_decorations
);
```

В C++ нет классов, похожих на `boost::dll::shared_library`. Но работа продолжается, и однажды мы, возможно, увидим их в стандарте C++.

См. также

Официальная документация содержит множество примеров и, что более важно, типичные проблемы и примеры неправильного использования: http://boost.org/libs/dll.

Получение backtrace – текущей последовательности вызовов

В сообщениях об ошибках и сбоях зачастую важнее сообщать о шагах, которые приводят к ошибке, а не о самой ошибке. Рассмотрим какой-то торговый симулятор:

```
int main() {
    int money = 1000;
    start_trading(money);
}
```

Все, что он сообщает, – это строка:

```
Sorry, you're bankrupt!
```

Так не пойдет. Мы хотим знать, как это произошло, какие шаги привели к банкротству!

Ладно. Давайте исправим следующую функцию и улучшим диагностическое сообщение:

```
void report_bankruptcy() {
    std::cout << "Sorry, you're bankrupt!\n";

    std::exit(0);
}
```

Приступим

Вам понадобится Boost версии 1.65 или выше для этого рецепта. Базовые знания C++ также являются обязательными.

Как это делается...

Для этого рецепта нам нужно будет только создать один класс и вывести его:

```
#include <iostream>
#include <boost/stacktrace.hpp>

void report_bankruptcy() {
   std::cout << "Sorry, you're bankrupt!\n";
   std::cout << "Here's how it happened:\n"
       << boost::stacktrace::stacktrace();

   std::exit(0);
}
```

Готово. Теперь функция `report_bankruptcy()` выводит что-то похожее на это (читайте снизу вверх):

```
Sorry, you're bankrupt!
Here's how it happened:
0# report_bankruptcy()
1# loose(int)
2# go_to_casino(int)
3# go_to_bar(int)
4# win(int)
5# go_to_casino(int)
6# go_to_bar(int)
7# win(int)
8# make_a_bet(int)
9# loose(int)
10# make_a_bet(int)
11# loose(int)
12# make_a_bet(int)
13# start_trading(int)
14# main
15# 0x00007F79D4C48F45 in /lib/x86_64-linuxgnu/

libc.so.6

16# 0x0000000000401F39 in ./04_stacktrace
```

Как это работает...

Вся магия находится в классе `boost::stacktrace::stacktrace`. При конструировании он быстро сохраняет текущий стек вызовов в себе. `boost::stacktrace::stacktrace` – копируемый и перемещаемый класс, поэтому сохраненную последовательность вызовов можно передавать другим функциям, копировать в классы исключений и даже хранить в каком-то файле. Делайте с ним, что хотите!

Экземпляры этого класса при выводе в поток декодируют сохраненную последовательность вызовов и пытаются получить понятные для человеческого восприятия имена функций. Это как раз то, что вы видели в предыдущем примере: последовательность вызовов, которая приводит к вызову функции `report_bankruptcy()`.

`boost::stacktrace::stacktrace` позволяет итерироваться по сохраненным адресам функций, декодировать отдельные адреса в понятные для человеческого восприятия имена. Если вам не нравится формат вывода по умолчанию, можете написать собственную функцию, которая выводит данные так, как вы предпочитаете.

Обратите внимание на то, что полезность обратной трассировки зависит от нескольких факторов. Оптимизированные сборки вашей программы могут использовать встраивание функций, что приводит к появлению менее читабельных трассировок:

```
0# report_bankruptcy()
1# go_to_casino(int)
2# win(int)
3# make_a_bet(int)
4# make_a_bet(int)
5# make_a_bet(int)
6# main
```

Сборка вашего исполняемого файла без отладочных символов может привести к появлению трассировки, где не будет большинства имен функций.

Прочтите раздел «*Конфигурация и сборка*» из официальной документации для получения дополнительной информации о различных флагах компиляции и макросах, которые могут повлиять на читабельность трассировки.

Дополнительно...

Библиотека `Boost.Stacktrace` имеет очень удобный для больших проектов функционал. Вы можете отключить всю трассировку при линковке вашей программы. Это означает, что вам не нужно заново собирать все свои исходные файлы. Просто определите макрос `BOOST_STACKTRACE_LINK` для всего проекта. Теперь, если вы линкуетесь к библиотеке `boost_stacktrace_noop`, трассировки будут пустые. Линкуйтесь с `boost_stacktrace_windbg/boost_stacktrace_windbg_cached/boost_stacktrace_backtrace/ ...`, чтобы получить трассировки различной степени читабельности.

`Boost.Stacktrace` – новая библиотека; она появилась в Boost версии 1.65.

Класс `boost::stacktrac::stacktrace` собирает последовательности текущих вызовов довольно быстро; он просто динамически выделяет фрагмент памяти и копирует в него кучу адресов.

Расшифровка адресов идет намного медленнее; здесь используется множество платформоспецифичных вызовов, могут создаваться процессы-потомки, инициализироваться и использоваться **COM**-объекты.

C++20 не имеет функциональности `Boost.Stacktrace`. Продолжается работа по добавлению его в стандарт C++23.

См. также

Официальная документация по адресу http://boost.org/libs/stacktrace/ содержит описание всех возможностей `Boost.Stacktrace`.

Быстрая передача данных из одного процесса в другой

Иногда мы пишем программы, которые много общаются друг с другом. Когда программы запускаются на разных машинах, использование сокетов является наиболее распространенным методом передачи данных. Но если несколько процессов выполняются на одной машине, можно сделать лучше!

Давайте посмотрим, как сделать фрагмент памяти доступным из разных процессов, используя библиотеку `Boost.Interprocess`.

Подготовка

Для этого рецепта требуются базовые знания C++, а также знание атомарных переменных (обратитесь к разделу «*См. также*» для получения дополнительной информации об атомарных операциях). Некоторые платформы требуют линковки с библиотекой времени выполнения `rt`.

Как это делается...

В этом примере мы будем использовать одну атомарную переменную из разных процессов, увеличивая ее при запуске нового процесса и уменьшая при завершении.

1. Нам нужно подключить следующий заголовочный файл для межпроцессного взаимодействия:

```
#include <boost/interprocess/managed_shared_memory.hpp>
```

2. Проверка поможет нам убедиться, что атомарные операции пригодны для этого примера:

```
#include <boost/atomic.hpp>

typedef boost::atomic<int> atomic_t;
#if (BOOST_ATOMIC_INT_LOCK_FREE != 2)
#error "This code requires lock-free boost::atomic<int>"
#endif
```

3. Создаем или получаем общий сегмент памяти:

```
int main() {
    boost::interprocess::managed_shared_memory
        segment(boost::interprocess::open_or_create, "shm1-cache", 1024);
```

4. Получаем или создаем переменную atomic:

```
atomic_t& atomic
    = *segment.find_or_construct<atomic_t> // 1
        ("shm1-counter") // 2
        (0) // 3
;
```

5. Работаем с переменной atomic обычным способом:

```
std::cout << "I have index " << ++ atomic
    << ". Press any key...\n";
std::cin.get();
```

6. Уничтожаем переменную:

```
    const int snapshot = --atomic;
    if (!snapshot) {
        segment.destroy<atomic_t>("shm1-counter");
        boost::interprocess::shared_memory_object
                ::remove("shm1-cache");
    }
} /*main*/
```

Вот и все! Теперь, если запустим несколько экземпляров этой программы одновременно, мы видим, что каждый новый экземпляр увеличивает свое индексное значение:

```
I have index 1. Press any key...
I have index 2.

Press any key...
I have index 3. Press any key...
I have index 4. Press any key...
I have index 5.

Press any key...
```

Как это работает...

Основная идея этого рецепта – получить сегмент памяти, видимый для всех процессов, и поместить в него некие данные. Давайте посмотрим на *этап 3*, где мы извлекаем такой сегмент памяти. Здесь "shm1-cache" – это имя сегмента. Мы можем создавать разные сегменты с разными именами. Первый параметр – boost::interprocess::open_or_create, который сообщает, что boost::interprocess::managed_shared_memory должен открыть существующий сегмент с именем shm1-cache или создать его. Последний параметр – это размер сегмента.

Размер сегмента должен быть достаточно большим, чтобы вместить вспомогательные данные библиотеки Boost.Interprocess. Вот почему мы использовали 1024, а не sizeof(atomic_t). Но на самом деле операционная система округляет это значение до ближайшего поддерживаемого значения, которое обычно равно или больше 4 килобайт.

Этап 4 сложный, поскольку здесь мы выполняем несколько задач одновременно. Во второй части этого этапа мы находим или создаем переменную с именем `"shm1-counter"` в сегменте. В `// 3` мы предоставляем параметр, который используется для инициализации переменной, если она не была найдена на `// 2`. Этот параметр используется, только если переменная не найдена и должна быть создана, иначе он игнорируется. Присмотритесь ко второй строке `// 1`. Видите вызов оператора разыменования `*`? Мы делаем это потому, что `segment.find_or_construct<atomic_t>` возвращает указатель на `atomic_t`, а работать с голыми указателями в C++ – плохой стиль.

Мы используем атомарные переменные в общей памяти! Это необходимо, потому что два или более процесса могут одновременно работать с одной и той же переменной `shm1-counter`.

Вы должны быть очень осторожны при работе с объектами в общей памяти; не забудьте уничтожить их! На *этапе 6* последний завершающийся процесс уничтожит объект и сегмент, используя их имена.

Дополнительно...

Внимательно посмотрите на этап 2, где мы проверяем, что `BOOST_ATOMIC_INT_LOCK_FREE` равен `2` и что `atomic_t` не использует мьютексы. Это очень важно, потому что обычные мьютексы не работают в общей памяти. Поэтому, если `BOOST_ATOMIC_INT_LOCK_FREE` не равен 2, мы получаем неопределенное поведение.

К сожалению, в C++11 нет классов для межпроцессных операций, и, насколько мне известно, `Boost.Interprocess` не предлагается включать в стандарт C++.

После создания сегмента он не может увеличиваться в размере автоматически! Убедитесь, что вы создаете достаточно большие сегменты для своих нужд, или посмотрите информацию в разделе «*См. также*» о ручном увеличении размеров сегментов.

Общая память является самым быстрым и весьма опасным способом взаимодействия процессов. Процессы должны выполняться на одном хосте или SMP-кластере, чтобы можно было воспользоваться общей памятью.

См. также

- Рецепт «*Синхронизация межпроцессного взаимодействия*» расскажет вам больше об общей памяти, межпроцессном обмене данными и синхронизации доступа к ресурсам в общей памяти;
- рецепт «*Быстрый доступ к общему ресурсу с использованием атомарных операций*» расскажет вам об атомарных операциях;
- официальная документация по `Boost.Interprocess` также может помочь; она доступна по адресу http://boost.org/libs/interprocess;
- как увеличить управляемые сегменты, описано в разделе *Growing managed segments* на странице http://boost.org/libs/interprocess.

Синхронизация межпроцессного взаимодействия

В предыдущем рецепте мы увидели, как создать общую память и как разместить в ней объекты. Теперь пришло время сделать что-то полезное. Давайте возьмем пример из рецепта «*Создание класса work_queue*» главы 5 «*Многопоточность*» и сделаем так, чтобы он работал для нескольких процессов. В конце этого примера мы получим класс, который может хранить различные задачи и передавать их между процессами.

Подготовка

Этот рецепт использует методы из предыдущего рецепта. Вам также нужно будет прочитать рецепт «*Создание класса work_queue*» главы 5 «*Многопоточность*», чтобы понять суть. Для этого примера на некоторых платформах требуется линковка с библиотекой `rt`.

Как это делается...

Бытует предположение, что порождение отдельных подпроцессов вместо потоков делает программу более надежной, потому что завершение работы подпроцесса не завершает работу основного процесса. Мы не будем спорить с этим предположением, а просто посмотрим, как можно реализовать обмен данными между процессами.

1. Для этого рецепта требуется много заголовочных файлов:

```
#include <boost/interprocess/managed_shared_memory.hpp>
#include <boost/interprocess/containers/deque.hpp>
#include <boost/interprocess/allocators/allocator.hpp>
#include <boost/interprocess/sync/interprocess_mutex.hpp>
#include <boost/interprocess/sync/interprocess_condition.hpp>
#include <boost/interprocess/sync/scoped_lock.hpp>

#include <boost/optional.hpp>
```

2. Теперь нам нужно определить нашу структуру `task_structure`, которая будет использоваться для хранения задач:

```
struct task_structure {
    // ...
};
```

3. Приступаем к написанию класса `work_queue`:

```
class work_queue {
public:
    typedef boost::interprocess::managed_shared_memory
            managed_shared_memory_t;
    typedef task_structure task_type;
    typedef boost::interprocess::allocator<
        task_type,
        boost::interprocess::managed_shared_memory::segment_manager
    > allocator_t;
```

4. Описывем члены `work_queue` следующим образом:

```
private:
    managed_shared_memory_t segment_;
    const allocator_t allocator_;

    typedef boost::interprocess::deque<task_type, allocator_t> deque_t;
    deque_t& tasks_;

    typedef boost::interprocess::interprocess_mutex mutex_t;
    mutex_t& mutex_;

    typedef boost::interprocess::interprocess_condition condition_t;
    condition_t& cond_;

    typedef boost::interprocess::scoped_lock<mutex_t>
scoped_lock_t;
```

5. Инициализация членов должна выглядеть так:

```
public:
    explicit work_queue()
        : segment_(
            boost::interprocess::open_or_create,
            "work-queue",
            1024 * 1024 * 32
        )
        , allocator_(segment_.get_segment_manager())
        , tasks_(
            *segment_.find_or_construct<deque_t>
              ("work-queue:deque")(allocator_)
        )
        , mutex_(
            *segment_.find_or_construct<mutex_t>
              ("work-queue:mutex")()
        )
        , cond_(
            *segment_.find_or_construct<condition_t>
              ("work-queue:condition")()
        )
{}
```

6. Нам нужно внести некоторые незначительные изменения в функции-члены, такие как использование scoped_lock_t:

```
boost::optional<task_type> try_pop_task() {
    boost::optional<task_type> ret;
    scoped_lock_t lock(mutex_);
    if (!tasks_.empty()) {
        ret = tasks_.front();
        tasks_.pop_front();
    }
    return ret;
}
```

7. Не забудьте про очистку ресурсов:

```
    void cleanup() {
        segment_.destroy<condition_t>("work-queue:condition");
        segment_.destroy<mutex_t>("work-queue:mutex");
        segment_.destroy<deque_t>("work-queue:deque");

        boost::interprocess::shared_memory_object
            ::remove("work-queue");
    }
```

Как это работает...

В этом рецепте мы делаем почти то же самое, что и в рецепте «*Создание класса work_queue*» главы 5 «*Многопоточность*», но располагаем данные в общей памяти.

Соблюдайте особую осторожность при хранении в общей памяти объектов, которые имеют указатели или ссылки в качестве полей-членов. Мы увидим, как справиться с указателями, в следующем рецепте.

Посмотрите на *этап 2*. Мы не использовали `boost::function` в качестве типа задачи, потому что в нем есть указатели, из-за чего он не работает в общей памяти.

Этап 3 интересен благодаря `allocator_t`. Этот аллокатор выделяет участки из сегмента общей памяти, доступной и другим процессам. `allocator_t` – это аллокатор с состоянием, он хранит информацию о сегменте общей памяти, и его внутреннее состояние копируется вместе с контейнером. Кроме того, у аллокатора отсутствует конструктор по умолчанию.

Этап 4 довольно прост, за исключением того, что у нас есть только ссылки на `tasks_`, `mutex_` и `cond_`. Это сделано потому, что сами объекты создаются в общей памяти. Таким образом, `work_queue` может хранить только ссылки, а не сами объекты.

На *этапе 5* мы инициализируем члены. Этот код должен быть вам знаком. Мы делали то же самое в предыдущем рецепте.

Мы предоставляем экземпляр аллокатора для `tasks_` при его создании, потому что `allocator_t` не может быть создан самим контейнером – внутреннее состояние аллокатора должно быть проинициализировано и хранить информацию о сегменте. Общая память не разрушается при завершении процесса, поэтому мы можем запустить программу один раз, разместить задачи в рабочей очереди, остановить программу, запустить какую-либо другую программу и получить задачи, сохраненные первым экземпляром программы. Общая память уничтожается только при перезапуске или явном вызове `segment.deallocate("work-queue");`.

Дополнительно...

Как уже упоминалось в предыдущем рецепте, в C++ нет классов из `Boost.Interprocess`. Более того, вы не должны использовать контейнеры C++20 или C++03

в сегментах общей памяти. Некоторые из этих контейнеров могут и заработать, но это поведение не является переносимым.

Если вы заглянете внутрь некоторых заголовочных файлов `<boost/interprocess/containers/*.hpp>`, то обнаружите, что они используют контейнеры из библиотеки `Boost.Containers`:

```
namespace boost { namespace interprocess {
    using boost::container::vector;
}}
```

Контейнеры `Boost.Interprocess` обладают всеми преимуществами библиотеки `Boost.Containers`, включая rvalue-ссылки и их эмуляцию на старых компиляторах.

`Boost.Interprocess` – самое быстрое решение для общения процессов, запущенных на одной машине.

См. также

- Рецепт «*Использование указателей в общей памяти*»;
- прочтите главу 5 «*Многопоточность*» для получения дополнительной информации о примитивах синхронизации и многопоточности;
- обратитесь к официальной документации по библиотеке `Boost.Interprocess` для получения дополнительных примеров и информации; она доступна по ссылке: https://www.boost.org/doc/libs/1_72_0/doc/html/interprocess.html.

Использование указателей в общей памяти

Трудно представить себе написание некоторых низкоуровневых базовых классов C++ без указателей. Указатели и ссылки в C++ повсюду, и они не работают в общей памяти! Таким образом, если у нас есть структура в общей памяти, и мы в одно из ее полей записываем адрес некой переменной из общей памяти, то этот адрес будет недействительным в другом процессе:

```
struct with_pointer {
    int* pointer_;  // так работать не будет
    // ...
    int value_holder_;
};
```

Как это исправить?

Подготовка

Для понимания этого рецепта необходимо прочитать предыдущий рецепт. Для этого примера на некоторых платформах требуется линковка с системной библиотекой `rt`.

Как это делается...

Исправить структуру очень просто; нам нужно только заменить указатель на `offset_ptr<int>`:

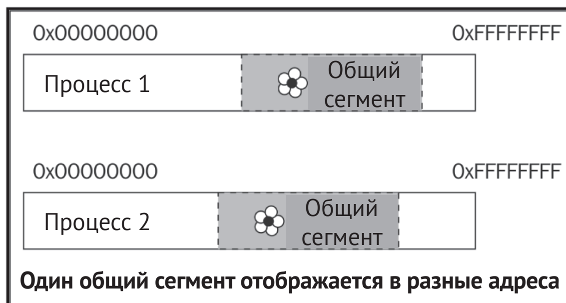
```
#include <boost/interprocess/offset_ptr.hpp>

struct correct_struct {
    boost::interprocess::offset_ptr<int> pointer_;
    // ...
    int value_holder_;
};
```

Теперь мы можем использовать его как обычный указатель:

```
int main() {
    boost::interprocess::managed_shared_memory
        segment(boost::interprocess::open_or_create, "segment", 4096);

    correct_struct* ptr =
        segment.find<correct_struct>("structure").first;

    if (ptr) {
        std::cout << "Structure found\n";
        assert(*ptr->pointer_ == ethalon_value);
        segment.destroy<correct_struct>("structure");
    }
}
```

Как это работает...

Мы не можем использовать указатели в общей памяти, потому что, когда часть этой памяти отображается в адресное пространство процесса, ее адрес действителен только для этого процесса. Когда мы получаем адрес переменной, это локальный для данного процесса адрес. Другие процессы будут отображать общую память в другой адрес, и в результате адрес переменной будет отличаться.



Один общий сегмент отображается в разные адреса

Итак, как же работать с адресом, который постоянно меняется? Есть одна хитрость! Поскольку указатель и структура находятся в одном и том же сегменте общей памяти, расстояние между ними не меняется. Идея `boost::interprocess::offset_ptr` заключается в том, чтобы запомнить это расстояние между адресом `offset_ptr` и указанным значением. При доступе `offset_ptr` добавляет значение расстояния к зависимому от процесса адресу переменной `offset_ptr`.

Указатель `offset_ptr` имитирует интерфейс сырых указателей, поэтому им легко воспользоваться для быстрой адаптации кода к работе в общей памяти.

Не помещайте классы, которые могут иметь указатели или ссылки, в общую память!

Дополнительно...

Указатель `offset_ptr` работает немного медленнее, чем обычный указатель, потому что при каждом разыменовании требуется вычислять адрес. Но эта разница обычно не должна вас беспокоить.

В C++ нет `offset_ptr` указателей.

См. также

- Официальная документация содержит множество примеров и описаний более продвинутой функциональности `Boost.Interprocess`; она доступна по адресу http://boost.org/libs/interprocess;
- рецепт «*Самый быстрый способ чтения файлов*» содержит информацию о нетипичном использовании библиотеки `Boost.Interprocess`.

Самый быстрый способ чтения файлов

В интернете люди спрашивают: «Какой самый быстрый способ чтения файлов?» Давайте усложним задачу для этого рецепта: какой самый быстрый и переносимый способ чтения двоичных файлов?

Подготовка

Для этого рецепта требуются базовые знания C++ и `std::fstream`.

Как это делается...

Техника из этого рецепта широко используется приложениями, чувствительными к производительности ввода-вывода. Это самый быстрый способ чтения файлов.

1. Нам нужно подключить два заголовка из библиотеки `Boost.Interprocess`:

```
#include <boost/interprocess/file_mapping.hpp>
#include <boost/interprocess/mapped_region.hpp>
```

2. Теперь нужно открыть файл:

```
const boost::interprocess::mode_t mode = boost::interprocess::read_only;
boost::interprocess::file_mapping fm(filename, mode);
```

3. Основная часть этого рецепта отображает все файлы в память:

```
boost::interprocess::mapped_region region(fm, mode, 0, 0);
```

4. Получение указателя на данные в файле:

```
const char* begin = static_cast<const char*>(

    region.get_address()
);
```

Готово! Теперь мы можем работать с файлом, как с обычной памятью:

```
const char* pos = std::find(
    begin, begin + region.get_size(), '\1'
);
```

Как это работает...

Все популярные операционные системы имеют возможность отображать файл в адресное пространство процессов. После того как такое отображение было выполнено, процесс может работать с этими адресами так же, как с обычной памятью. Операционная система сама заботится обо всех файловых операциях, таких как кеширование и упреждающее чтение.

Почему это быстрее, чем традиционные операции чтения и записи? Это связано с тем, что в большинстве случаев чтение и запись реализуются как отображение в память и копирование данных в указанный пользователем буфер. Таким образом, чтение обычно делает немного больше, чем отображение.

Как и в случае с `std::fstream` из стандартной библиотеки, мы должны передать режим открытия файла. *См. этап 2*, где мы предоставили режим `boost::interprocess::read_only`.

См. этап 3, где мы отобразили весь файл сразу. Эта операция действительно очень быстрая, потому что ОС не читает все данные с диска, а сразу возвращает нам управление и ожидает запросов к части отображаемой области. После запроса ОС загружает запрошенную часть файла с диска в память. Как мы видим, операции отображения в память являются ленивыми, а размер отображаемой области не влияет на производительность.

Однако 32-разрядная ОС не может отображать в память большие файлы, поэтому вам придется отображать их по частям. Операционные системы POSIX (Linux) требуют определения макроса `_FILE_OFFSET_BITS=64` для всего проекта, чтобы работать с большими файлами на 32-битной платформе. В противном случае ОС не сможет отобразить части файла, размер находится за границей первых 4 ГБ.

Теперь пришло время измерить производительность:

```
$ TIME="%E" time ./reading_files m
mapped_region: 0:00.08
$ TIME="%E" time ./reading_files r
ifstream: 0:00.09
$ TIME="%E" time ./reading_files a
C: 0:00.09
```

Как и ожидалось, отображенные в память файлы немного быстрее по сравнению с традиционными операциями чтения. Также видно, что чистые методы C имеют такую же производительность, что и класс C++ `std::ifstream`, поэтому по возможности не используйте `FILE*` функции в C++. Они предназначены только для C, а не для C++!

Для обеспечения оптимальной производительности `std::ifstream` не забудьте открыть файлы в двоичном режиме и читать данные блоками:

```
std::ifstream f(filename, std::ifstream::binary);
// ...
char c[kilobyte];
f.read(c, kilobyte);
```

Дополнительно...

К сожалению, классы для отображения файлов в память не являются частью C++20, и, похоже, их не будет и в C++23.

Запись в области с отображением в память также является очень быстрой операцией. Операционная система кеширует записи и не сбрасывает изменения на диск немедленно. Существует разница между кешированием данных в ОС и `std::ofstream`. В случае `std::ofstream` данные кешируются приложением, и если работа приложения аварийно завершается, закешированные данные могут быть потеряны. Когда данные кешируются ОС, завершение работы приложения не приводит к их потере. Сбои питания и сбои ОС приводят к потере данных в обоих случаях.

Если несколько процессов отображают один файл и один из процессов изменяет отображаемую область, то изменения сразу видны другим процессам (даже без фактической записи данных на диск! Современные ОС очень умные! Только не забывайте про синхронизацию доступа).

См. также

Библиотека `Boost.Interprocess` содержит множество полезных функций для работы с системой; не все из них описаны в этой книге. Вы можете прочитать больше об этой грандиозной библиотеке на официальном сайте: http://boost.org/libs/interprocess.

Сопрограммы – сохранение состояния и откладывание выполнения

В настоящее время множество встроенных устройств по-прежнему имеют только одно ядро. Разработчики пишут код для этих устройств, пытаясь выжать из них максимальную производительность.

Использование `Boost.Threads` или другой библиотеки потоков для таких устройств неэффективно. Операционная система будет вынуждена тратить ресурсы процессора на планирование потоков выполнения, управлять ресурсами и т. д., при этом оборудование не может выполнять потоки параллельно.

Итак, как заставить программу переключиться на выполнение подпрограммы, пока ожидается какой-то ресурс в основной части программы? Более того, как контролировать время выполнения подпрограммы?

Подготовка

Для этого рецепта требуются базовые знания C++ и шаблонов. Чтение рецептов о `Boost.Function` также может помочь.

Как это делается...

Этот рецепт касается **сопрограмм** или **подпрограмм**, которые допускают несколько точек входа. Несколько точек входа дают нам возможность приостановить и возобновить выполнение программы в определенных местах, переключаясь на другие подпрограммы.

1. Библиотека `Boost.Coroutine2` позаботится практически обо всем. Нам просто нужно подключить ее заголовочный файл:

```
#include <boost/coroutine2/coroutine.hpp>
```

2. Создайте тип сопрограммы с требуемым типом входного параметра:

```
typedef boost::coroutines2::asymmetric_coroutine<std::size_t> corout_t;
```

3. Создайте класс, представляющий подпрограмму:

```
struct coroutine_task {
    std::string& result;

    coroutine_task(std::string& r)
        : result(r)
    {}

    void operator()(corout_t::pull_type& yield);
private:
    std::size_t ticks_to_work;
    void tick(corout_t::pull_type& yield);
};
```

4. Давайте создадим сопрограмму:

```
int main() {
    std::string result;
    coroutine_task task(result);
    corout_t::push_type coroutine(task);
```

5. Теперь мы можем выполнить подпрограмму, ожидая какого-то события в главной программе:

```
// Где-то в функции main():

while (!spinlock.try_lock()) {
    // Мы можем выполнять какую-то полезную работу, прежде чем попытаться захватить
    // spinlock   еще раз.
    coroutine(10); // 10 - количество тиков для работы.
}
// `spinlock` захвачен.
// ...

while (!port.block_ready()) {
   // Мы можем выполнять какую-то полезную работу, прежде чем получить блок данных
   // еще раз.
   coroutine(300); // 300 - количество тиков для работы.

   // Выполняем какие-то действия с переменной `result`.
}
```

6. Метод coroutine может выглядеть так:

```
void coroutine_task::operator()(corout_t::pull_type& yield) {
    ticks_to_work = yield.get();

    // Подготовка буферов.
    std::string buffer0;

    while (1) {
        const bool requiers_1_more_copy = copy_to_buffer(buffer0);
        tick(yield);

        if (requiers_1_more_copy) {
            std::string buffer1;
            copy_to_buffer(buffer1);
            tick(yield);
            process(buffer1);
            tick(yield);
        }

        process(buffer0);
        tick(yield);
     }
}
```

7. Функцию tick() можно реализовать так:

```
void coroutine_task::tick(corout_t::pull_type& yield) {
    if (ticks_to_work != 0) {
        --ticks_to_work;
    }

    if (ticks_to_work == 0) {
        // Переключаемся обратно.
        yield();

        ticks_to_work = yield.get();
    }
}
```

Как это работает...

На *этапе 2* мы описываем входной параметр нашей подпрограммы, используя std::size_t в качестве параметра шаблона. В нашем примере этот параметр будет обозначать некое условное время, отведенное функции на работу, – «тики» (англ. ticks).

Этап 3 довольно прост, за исключением параметров corout_t::pull_type& yield. Мы увидим его в действии чуть ниже.

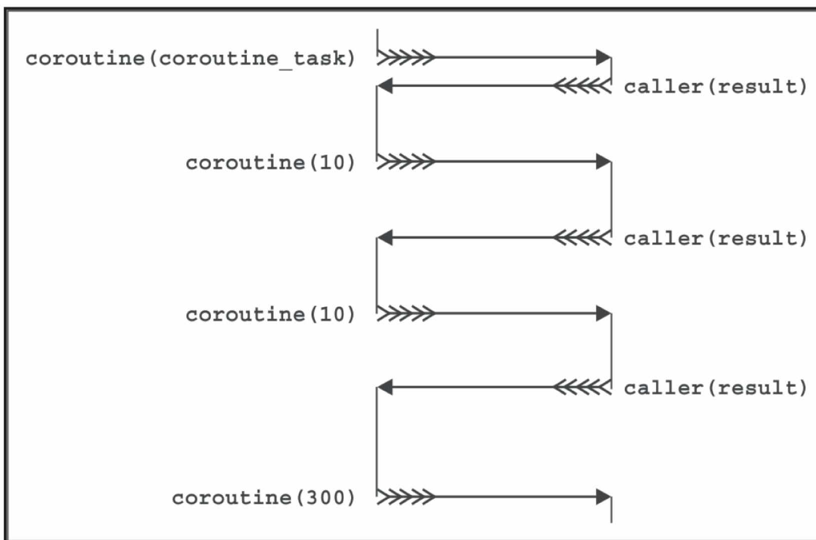
Когда мы вызываем coroutine(10) на *этапе 5*, то запускаем выполнение сопрограммы. Выполнение переходит к coroutine_task::operator(), где вызов yield.get() возвращает входной параметр 10. Выполнение продолжается, а функция coroutine_task::tick измеряет затраченное время в тиках.

Теперь самое интересное!

На этапе 7, если в функции coroutine_task::tick переменная ticks_to_work становится равной 0, выполнение сопрограммы приостанавливается в функции yield(), а функция main() продолжает выполнение. При следующем вызове coroutine(some_value) выполнение сопрограммы продолжается с середины функции tick, прямо на строке рядом с функцией yield(). В этой строке выполняется ticks_to_work = yield.get();, в переменной ticks_to_work появляется новое входное значение some_value.

Это означает, что мы можем приостановить/продолжить сопрограмму в нескольких местах внутри функции. Все состояние функции и переменные восстанавливаются.

Позвольте мне описать основное различие между сопрограммами и потоками. Когда выполняется сопрограмма, главная задача ничего не делает. Когда главная задача выполняется, задача сопрограммы ничего не делает. В случае с потоками такой гарантии нет. В сопрограммах вы явно указываете, когда начинать подзадачу, а когда приостанавливать ее. Даже в одноядерной среде потоки могут переключаться в любой момент; их поведение нельзя контролировать.



Дополнительно...

При переключении потоков ОС выполняет большую работу, поэтому это не очень быстрая операция. Однако с сопрограммами у вас есть полный контроль над переключениями задач; более того, вам не нужно выполнять какую-то специфическую для ОС работу при переходе в режим ядра. Переключение сопрограмм происходит намного быстрее, чем переключение потоков, но не так быстро, как вызов boost::function.

Библиотека Boost.Coroutine2 заботится о вызове деструктора для переменных, созданных в задаче сопрограммы, поэтому не нужно беспокоиться об утечках.

Сопрограммы используют исключение `boost::coroutines2::detail::force_unwind` для освобождения ресурсов. Это исключение не наследуется от `std::exception`. Старайтесь не перехватывать это исключение в задачах сопрограммы.

Копировать сопрограммы `Boost.Coroutine2` нельзя, но их можно перемещать, используя `std::move`.

Существует библиотека `Boost.Coroutine` (без цифры 2 в конце!), для которой не требуется компилятор, совместимый с C++11. Но эта библиотека устарела, и она имеет некоторые отличия (например, она не выкидывает исключения из сопрограмм). Имейте это в виду при разработке! `Boost.Coroutine` также значительно изменила свой интерфейс в Boost версии 1.56.

В C++17 нет сопрограмм. Но сопрограммы есть в стандарте C++20.

C++20-корутины отличаются от `Boost.Coroutine2`! Boost предоставляет **стековые** (англ. **stackful**) сопрограммы, а это означает, что вам не нужно специально декорировать свой код макросами или ключевыми словами, чтобы использовать их. Но это также означает, что сопрограммы Boost сложнее оптимизировать компилятору и что они могут выделять больше памяти. Корутины C++20 предоставляют **бесстековые (англ. stackless)** сопрограммы. Это означает, что компилятор может точно вычислить количество необходимой памяти для сопрограммы и даже оптимизировать всю сопрограмму.

Однако этот подход требует изменений кода, и под него может быть немного сложнее адаптировать имеющийся код.

См. также

- Официальная документация Boost содержит дополнительные примеры, заметки о производительности, ограничения и варианты использования для библиотеки `Boost.Coroutines2`; она доступна по следующей ссылке: http://boost.org/libs/coroutine2;
- взгляните на рецепты главы 2 «*Управление ресурсами*» и главы 5 «*Многопоточность*», чтобы узнать разницу между библиотеками `Boost.Coroutine`, `Boost.Thread` и `Boost.Function`;
- интересуют корутины в C++20? Вот интересная беседа, касающаяся реализации, от автора *CppCon 2016: Gor Nishanov. C++ Coroutines: Under the covers*: https://www.youtube.com/watch?v=8C8NnE1Dg4A.

Глава 12

Касаясь верхушки айсберга

Темы, которые мы рассмотрим в этой главе:

- работа с графами;
- визуализация графов;
- использование генератора истинно случайных чисел;
- использование переносимых математических функций;
- написание тестовых случаев;
- объединение нескольких тестовых случаев в одном тестовом модуле;
- манипулирование изображениями.

Вступление

Boost – это огромная коллекция библиотек. Некоторые из этих библиотек небольшие и предназначены для повседневного использования, в то время как для других требуется отдельная книга, чтобы описать все их возможности.

Эта глава посвящена некоторым из этих больших библиотек и дает общее представление о них.

В первых двух рецептах объясняется использование `Boost.Graph`. Это большая библиотека с безумным количеством алгоритмов. Мы рассмотрим основы и, возможно, самую важную часть разработки – визуализацию графов.

Мы также увидим очень полезный рецепт для генерации истинно случайных чисел. Это очень важный рецепт для написания безопасных криптографических систем.

В некоторых стандартных библиотеках C++ отсутствуют математические функции. Посмотрим, как можно это исправить с помощью Boost.

Для написания любой качественной системы вам понадобятся знания из рецептов «*Написание тестовых случаев*» и «*Объединение нескольких тестовых случаев в одном тестовом модуле*».

Последний рецепт касается библиотеки, которая помогла мне во многих курсовых работах во время учебы в университете. С ее помощью можно создавать и изменять изображения. Лично я использовал ее для визуализации различных алгоритмов, сокрытия данных в изображениях и создания текстур.

К сожалению, даже эта глава не может рассказать вам обо всех библиотеках Boost. Может быть, когда-нибудь я напишу еще одну книгу, а потом еще несколько.

Работа с графами

Некоторые задачи требуют представления данных в виде графа. Boost.Graph – это библиотека, разработанная для обеспечения гибкого способа построения и представления графов в памяти. Она также содержит множество алгоритмов для работы с графами, таких как топологическая сортировка, поиск в ширину, поиск в глубину и алгоритм Дейкстры.

Что же, давайте решим какие-нибудь базовые задачи с помощью этой библиотеки!

Подготовка

Для этого рецепта требуются только базовые знания C++ и шаблонов.

Как это делается...

В этом рецепте мы опишем тип графа, создадим граф заданного типа, добавим к нему несколько вершин и ребер и произведем поиск вершины. Этого должно быть достаточно, чтобы начать работу с Boost.Graph.

1. Начнем с описания типа графа:

```
#include <boost/graph/adjacency_list.hpp>
#include <string>

typedef std::string vertex_t;
typedef boost::adjacency_list<
    boost::vecS
    , boost::vecS
    , boost::bidirectionalS
    , vertex_t
> graph_type;
```

2. Теперь создадим его:

```
int main() {
    graph_type graph;
```

3. Давайте выполним недокументированный трюк, ускоряющий создание графа:

```
static const std::size_t vertex_count = 5;
graph.m_vertices.reserve(vertex_count);
```

4. Теперь мы готовы добавить вершины:

```
typedef boost::graph_traits<
    graph_type
>::vertex_descriptor descriptor_t;

descriptor_t cpp
    = boost::add_vertex(vertex_t("C++"), graph);
descriptor_t stl
    = boost::add_vertex(vertex_t("STL"), graph);
descriptor_t boost
    = boost::add_vertex(vertex_t("Boost"), graph);
```

```
descriptor_t guru
    = boost::add_vertex(vertex_t("C++ guru"), graph);
descriptor_t ansic
  = boost::add_vertex(vertex_t("C"), graph);
```

5. Настало время соединить вершины с ребрами:

```
   boost::add_edge(cpp, stl, graph);
   boost::add_edge(stl, boost, graph);
   boost::add_edge(boost, guru, graph);
   boost::add_edge(ansic, guru, graph);
} // Конец функции main()
```

6. Мы можем сделать функцию, которая ищет вершину:

```
inline void find_and_print(
    const graph_type& graph, boost::string_ref name)
{
```

7. Далее приведен код, который получает итераторы по всем вершинам:

```
typedef typename boost::graph_traits<
    graph_type
>::vertex_iterator vert_it_t;

vert_it_t it, end;
boost::tie(it, end) = boost::vertices(graph);
```

8. Пришло время выполнить поиск нужной вершины:

```
   typedef typename boost::graph_traits<
       graph_type
   >::vertex_descriptor desc_t;

   for (; it != end; ++it) {
       const desc_t desc = *it;
       const vertex_t& vertex = boost::get(
           boost::vertex_bundle, graph
       )[desc];

       if (vertex == name.data()) {
           break;
       }
   }

   assert(it != end);
   std::cout << name << '\n';
} /* find_and_print */
```

Как это работает...

На *этапе 1* мы описываем, как должен выглядеть наш граф и на каких типах он должен быть основан. `boost::adjacency_list` – это класс, который представляет графы в виде двумерных структур, где первое измерение содержит вершины, а второе измерение содержит ребра для этой вершины. `boost::adjacency_list` должен быть вашим выбором по умолчанию для представления графа, он подходит для большинства случаев.

Первый параметр шаблона boost::adjacency_list описывает структуру, используемую для представления списка ребер каждой из вершин. Второй – описывает структуру для хранения вершин. Мы можем выбрать разные контейнеры стандартной библиотеки для этих структур, используя предопределенные селекторы, как указано в приведенной ниже таблице:

Селектор	Контейнер стандартной библиотеки
boost::vecS	std::vector
boost::listS	std::list
boost::slistS	std::slist
boost::setS	std::set
boost::multisetS	std::multiset
boost::hash_setS	std::hash_set

Третий параметр шаблона используется для создания ненаправленного, направленного или двунаправленного графа. Используйте селекторы boost::undirectedS, boost::direS и boost::bidirectionalS соответственно.

Четвертый параметр шаблона описывает тип данных, который используется в качестве вершины. В нашем примере мы выбираем std::string. Мы также можем поддерживать тип данных для ребер и предоставлять его в качестве шестого параметра шаблона. В данном примере мы этого не делали.

Этапы 1 и *2* просты, но на *этапе 3* вы видите недокументированный способ ускорить построение графа. В нашем примере мы используем std::vector в качестве контейнера для хранения вершин, поэтому можем заставить его зарезервировать память для необходимого количества вершин. Это приводит к меньшему количеству выделений и освобождений памяти, меньшему количеству операций копирования при добавлении вершин в граф. Этот шаг не очень переносим и может перестать работать в одной из будущих версий Boost, поскольку сильно зависит от текущей реализации boost::adjacency_list и от выбранного типа контейнера для хранения вершин.

На *этапе 4* видно, как можно добавить вершины в граф. Обратите внимание, что был использован boost::graph_traits<graph_type>. Класс boost::graph_traits применяется для получения типов, специфичных для графа. Мы увидим его использование и описание некоторых специфичных для графов типов позже в этой главе. *Этап 5* показывает код, необходимый для соединения вершин ребрами.

Если бы мы предоставили некий тип данных для ребер, добавление ребра выглядело бы следующим образом: boost::add_edge(ansic, guru,edge_t(initialization_parameters), graph).

На *этапе 6* все тривиально. Однако в реальном проекте стоило бы сделать тип графа шаблонным параметром, чтобы обеспечить работу этой функции и с другими типами графов.

На этапе 7 мы видим, как перебирать все вершины графа. Тип итератора получен из boost::graph_traits. Функция boost::tie является частью библиотеки Boost.Tuple и используется для получения значений из кортежей

в переменные. Таким образом, вызов boost::tie(it, end) = boost::vertices(g) помещает итератор begin в переменную it, а итератор на конец диапазона вершин в переменную end.

Возможно, это вас удивит, но разыменование вершинного итератора не возвращает ссылку на вершину. Вместо этого он возвращает дескриптор вершины desc, который можно использовать в boost::get(boos::vertex_bundle, g)[desc] для получения ссылки на вершину, что мы и сделали на *этапе 8*. Тип дескриптора вершины используется во многих функциях Boost.Graph. Мы уже видели его применение в функции построения ребра на *этапе 5*.

Как уже упоминалось, библиотека Boost.Graph содержит реализации множества алгоритмов. Вы можете найти много политик поиска, но мы не будем обсуждать их в этой книге. Этот рецепт ограничивается только основами библиотеки графов.

Дополнительно...

Библиотека Boost.Graph не является частью C++20 и, скорее всего, не будет частью следующего стандарта C++. Текущая реализация не поддерживает возможности C++11, такие как rvalue-ссылки. Если вы используете вершины, у которых медленный конструктор копирования, можно ускорить код, используя следующий трюк:

```
vertex_descriptor desc = boost::add_vertex(graph);
boost::get(boost::vertex_bundle, g_)[desc] = std::move(vertex_data);
```

Он избегает копирования внутри boost::add_vertex(vertex_data, graph) и вместо этого использует конструктор по умолчанию с перемещением.

Эффективность Boost.Graph зависит от нескольких факторов, таких как базовые типы контейнеров, представление графа, типы ребер и вершин.

См. также

Чтение рецепта «*Визуализация графов*» упростит вам работу с графами. Вы также можете прочитать официальную документацию по следующей ссылке: http://boost.org/libs/graph.

Визуализация графов

Создание программ, которые манипулируют графами, никогда не было легким. Это связано с проблемами визуализации. Когда мы работаем с контейнерами стандартной библиотеки, такими как std::map и std::vector, мы всегда можем вывести содержимое контейнера и посмотреть, что происходит внутри. Но когда мы работаем со сложными графами, трудно наглядно визуализировать контент; текстовое представление не очень подходит для человеческого восприятия, потому что обычно оно содержит слишком много вершин и ребер.

В этом рецепте мы рассмотрим визуализацию графа с помощью утилит **Graphviz**.

Подготовка

Для визуализации графов вам понадобится инструмент визуализации Graphviz, а также знание предыдущего рецепта.

Как это делается...

Визуализация выполняется в два этапа. На первом этапе мы заставляем нашу программу выводить описание графа в текстовом формате, подходящем для Graphviz. На втором этапе мы импортируем выходные данные с первого шага в инструмент визуализации. Шаги, приведенные в этом рецепте, касаются первого этапа.

1. Давайте напишем оператор `std::ostream` для `graph_type`:

```
#include <boost/graph/graphviz.hpp>

std::ostream& operator<<(std::ostream& out, const graph_type& g) {
    detail::vertex_writer<graph_type> vw(g);
    boost::write_graphviz(out, g, vw);

    return out;
}
```

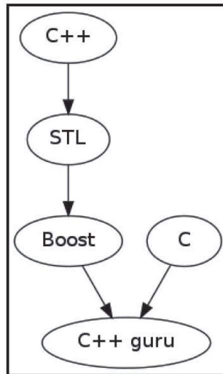
2. Структура `detail::vertex_writer`, используемая на предыдущем этапе, должна быть определена так:

```
#include <iosfwd>

namespace detail {
    template <class GraphT>
    class vertex_writer {
        const GraphT& g_;

    public:
        explicit vertex_writer(const GraphT& g)
            : g_(g)
        {}

        template <class VertexDescriptorT>
        void operator()(
            std::ostream& out,
            const VertexDescriptorT& d) const
        {
            out << " [label=\""
                << boost::get(boost::vertex_bundle, g_)[d]
                << "\"]";
        }
     }; // vertex_writer
 } // namespace detail
```

На этом все. Теперь, если мы выведем граф из предыдущего рецепта, используя команду `std::cout << graph;`, вывод можно будет использовать для создания графических изображений с использованием утилиты командной строки `dot`:

```
$ dot -Tpng -o dot.png
digraph G {
0 [label="C++"];
1 [label="STL"];
2 [label="Boost"];
3 [label="C++ guru"];
4 [label="C"];
0->1 ;
1->2 ;
2->3 ;
4->3 ;
}
```

Вывод предыдущей команды изображен на этом рисунке.



Можно также использовать программы **Gvedit** или **XDot** для визуализации, если вас пугает командная строка.

Как это работает...

Библиотека `Boost.Graph` содержит функцию для вывода графов в формате Graphviz (DOT-формат). Если мы напишем `boost::write_graphviz(out, g)` только с двумя параметрами на *этапе 1*, функция выведет лишь граф с вершинами, пронумерованными начиная с 0. Это не очень полезно, поэтому мы предоставляем экземпляр класса `vertex_writer`, который дополнительно выводит имена вершин.

Как видно на *этапе 2*, Graphviz понимает формат DOT. Если вы хотите вывести больше информации для своего графа, вам может потребоваться прочитать документацию Graphviz для получения дополнительной информации о формате DOT. Не бойтесь, он очень простой.

Если вы хотите добавить какие-то данные к ребрам во время визуализации, нужно предоставить экземпляр визуализатора ребер в качестве четвертого параметра `boost::write_graphviz`.

Дополнительно...

В C++ нет `Boost.Graph` или инструментов для визуализации графов. Существует множество других форматов графов и инструментов визуализации, и `Boost.Graph` может работать со многими из них.

См. также

- Рецепт «*Работа с графами*» содержит информацию о работе с библиотекой `Boost.Graphs`;
- множество информации о формате DOT и Graphviz можно найти на сайте http://www.graphviz.org;
- официальная документация по библиотеке `Boost.Graph` содержит множество примеров и полезной информации. Ее можно найти по адресу http://boost.org/libs/graph.

Использование генератора истинно случайных чисел

Я знаю много примеров коммерческих продуктов, которые используют неправильные методы для получения случайных чисел. Жаль, что некоторые компании по-прежнему применяют функцию `rand()` в криптографии и банковском программном обеспечении.

Давайте посмотрим, как получить полностью случайное **равномерное распределение**, используя библиотеку `Boost.Random`, которая лучше подходит для банковского программного обеспечения.

Приступаем

Для этого рецепта требуются базовые знания C++. Знание различных типов дистрибутивов также будет полезно. Код в этом рецепте требует линковки с библиотекой `boost_random`.

Как это делается...

Чтобы создать истинно случайное число, нам нужна помощь операционной системы или процессора. Вот как это можно сделать с помощью Boost.

1. Нам нужно подключить следующие заголовочные файлы:

```
#include <boost/config.hpp>
#include <boost/random/random_device.hpp>
#include <boost/random/uniform_int_distribution.hpp>
```

2. У поставщиков случайных битов разные имена для разных платформ:

```
int main() {
    static const std::string provider =
#ifdef BOOST_WINDOWS
        "Microsoft Strong Cryptographic Provider"
#else
        "/dev/urandom"
#endif
```

3. Теперь мы готовы инициализировать генератор с помощью `Boost.Random`:

```
boost::random_device device(provider);
```

4. Давайте получим равномерное распределение, которое возвращает значения между 1000 и 65 535:

```
boost::random::uniform_int_distribution<unsigned short> random(1000);
```

Вот и все. Теперь мы можем получать чисто случайные числа, используя вызов `random(device)`.

Как это работает...

Почему функция `rand()` не подходит, когда речь идет о банках? Она генерирует псевдослучайные числа, а значит, злоумышленник может предсказать следующее сгенерированное число. Это проблема всех алгоритмов псевдослучайных чисел. Некоторые алгоритмы проще прогнозировать, а некоторые сложнее, но все же возможно.

Вот почему в этом примере мы используем `boost::random_device` (*см. этап 3*). Это класс для работы с **энтропией** – информацией о случайных событиях, происходящих в вашем компьютере и заботливо собираемых ОС, для создания непредсказуемых равномерно распределенных случайных битов. Примеры таких событий – задержки между нажатиями клавиш клавиатуры, задержки между некоторыми аппаратными прерываниями и аппаратные генераторы случайных битов ЦП.

В операционных системах может быть несколько типов генераторов истинно случайных битов. В нашем примере для систем POSIX мы использовали `/dev/urandom` вместо более безопасного `/dev/random`, поскольку последний не возвращает управление вашей программе до тех пор, пока ОС не перехватит достаточное количество случайных событий. Ожидание энтропии может занять несколько секунд, что обычно не подходит для многих приложений. Используйте `/dev/random` для долговечных ключей **GPG/SSL/SSH**.

Теперь, когда мы закончили с генераторами, пришло время перейти к *этапу 4* и поговорить о классах распределения. Если генератор просто генерирует равномерно распределенные биты, то класс распределения создает случайное число из этих битов. На *этапе 4* мы создали равномерное распределение, которое возвращает случайное число типа `unsigned short`. Параметр `1000` означает, что распределение должно возвращать числа, большие или равные `1000`. В качестве второго параметра мы также можем указать максимальное число, которое по умолчанию равно максимальному значению, хранимому в возвращаемом типе.

Дополнительно...

В библиотеке `Boost.Random` имеется огромное количество генераторов истинно или псевдослучайных битов и распределений для различных нужд. Избегайте копирования распределений и генераторов. Это может оказаться дорогостоящей операцией.

В C++11 есть поддержка различных классов распределений и генераторов. Вы можете найти все классы из этого примера в заголовке `<random>` в пространстве имен `std::`. Библиотеки `Boost.Random` не используют возможности C++11, да они и не нужны для этой библиотеки. Стоит использовать реализацию Boost или стандартную библиотеку? Boost обеспечивает лучшую переносимость. Однако некоторые стандартные библиотеки имеют оптимизированные реализации и могут предоставлять полезные расширения.

См. также

Официальная документация содержит полный список генераторов и распределений с описаниями. Она доступна по этой ссылке: http://boost.org/libs/random.

Использование переносных математических функций

Некоторые проекты требуют тригонометрических функций, библиотек для численного решения обыкновенных дифференциальных уравнений, работы с распределениями и константами.

Все эти возможности `Boost.Math` будет сложно описать даже в отдельной книге. Одного рецепта точно не хватит. Поэтому давайте сосредоточимся на самых основных функциях для повседневного использования с типами с плавающей точкой.

Мы напишем переносимую функцию, которая проверяет входное значение на бесконечность и **Not-A-Number** (**NaN**) значений; меняет знак, если значение является отрицательным.

Подготовка

Для этого рецепта требуются базовые знания C++. Те, кто знает стандарт C99, найдут много знакомого в этом рецепте.

Как это делается...

1. Нам нужны следующие заголовочные файлы:

```
#include <boost/math/special_functions.hpp>
#include <cassert>
```

2. Проверку на бесконечность и NaN можно сделать так:

```
template <class T>
void check_float_inputs(T value) {
    assert(!boost::math::isinf(value));
    assert(!boost::math::isnan(value));
```

3. Используйте этот код для изменения знака:

```
    if (boost::math::signbit(value)) {
        value = boost::math::changesign(value);
    }

    // ...
} // check_float_inputs
```

Готово! Теперь можно убедиться, что наши проверки `assert` срабатывают на `check_float_inputs(std::sqrt(-1.0))` и `check_float_inputs (std::numeric_limits<double>::max() * 2.0)`.

Как это работает...

Вещественные типы имеют особые значения, которые нельзя проверить с помощью операторов равенства. Например, если переменная `v` содержит NaN, `assert(v != v)` может пройти, а может и не пройти в зависимости от компилятора.

Для таких случаев библиотека Boost.Math предоставляет функции, которые могут надежно проверять бесконечности и значения NaN.

Этап 3 содержит функцию boost::math::signbit, которая требует пояснения. Эта функция возвращает бит со знаком, который равен 1, если число отрицательное, и 0, если число положительное. Другими словами, она возвращает true, если значение отрицательное.

Глядя на *этап 3*, некоторые читатели могут спросить, почему нельзя просто умножить на -1 вместо вызова boost::math::changesign? Можно. Но умножение может работать медленнее, чем boost::math::changesign, и нет гарантий, что это будет работать для специальных значений. Например, если ваш код может работать с nan, код на *этапе 3* может изменить знак -nan и записать nan в переменную.

Лица, отвечающие за поддержку библиотеки Boost.Math, рекомендуют оборачивать математические функции из этого примера в круглые скобки, чтобы избежать коллизий с макросами C. Лучше написать (boost::math::isinf)(значение) вместо boost::math::isinf(значение).

Дополнительно…

В C99 есть все функции, описанные в этом рецепте. Зачем они нам нужны в Boost? Ну, некоторые производители компиляторов считают, что программистам не нужна полная поддержка C99, поэтому вы не найдете этих функций по крайней мере в одном очень популярном компиляторе. Еще одна причина состоит в том, что функции Boost.Math можно использовать и для классов, которые ведут себя как числа.

Boost.Math – очень быстрая, переносимая и надежная библиотека. **Математические специальные функции** являются частью библиотеки Boost.Math, и некоторые из них были приняты в C++17. Boost.Math, однако, предоставляет больше функций, и у нее есть очень удобные рекуррентные версии, которые имеют более подходящую сложность и лучше подходят для некоторых задач (таких как численные интегрирования).

См. также

Официальная документация Boost содержит множество интересных примеров и учебных материалов, которые помогут вам привыкнуть к Boost.Math. Перейдите по ссылке http://boost.org/libs/math, чтобы прочитать об этом.

Написание тестовых случаев

Этот и последующий рецепт посвящены автоматическому тестированию с использованием библиотеки Boost.Test, которая применяется для проверки функциональности многих библиотек Boost. Давайте напишем несколько тестов для нашего собственного класса:

```
#include <stdexcept>
struct foo {
    int val_;
```

```
    operator int() const;
    bool is_not_null() const;
    void throws() const; // Выбрасывает std::logic_error
};
```

Подготовка

Для этого рецепта требуются базовые знания C++. Чтобы скомпилировать код из этого рецепта, определите макрос `BOOST_TEST_DYN_LINK` и линкуйтесь с библиотеками `boost_unit_test_framework` и `boost_system`.

Как это делается...

Если честно, это не единственная библиотека для написания тестов в Boost, но самая функциональная.

1. Чтобы использовать ее, нам нужно определить макрос и подключить следующий заголовочный файл:

    ```
    #define BOOST_TEST_MODULE test_module_name
    #include <boost/test/unit_test.hpp>
    ```

2. Каждый набор тестов должен быть записан в тестовом случае (англ. test case):

    ```
    BOOST_AUTO_TEST_CASE(test_no_1) {
    ```

3. Проверка функции на результат `true` должна проводиться следующим образом:

    ```
    foo f1{1}, f2{2};
    BOOST_CHECK(f1.is_not_null());
    ```

4. Проверка на неравенство должна быть реализована так:

    ```
    BOOST_CHECK_NE(f1, f2);
    ```

5. Проверка на то, что выбрасывается исключение нужного типа, должно выглядеть следующим образом:

    ```
        BOOST_CHECK_THROW(f1.throws(), std::logic_error);
    } // BOOST_AUTO_TEST_CASE(test_no_1)
    ```

Готово! После компиляции и линковки у нас будет двоичный файл, который автоматически проверяет `foo` и выводит результаты теста в удобочитаемом формате.

Как это работает...

Писать модульные тесты легко. Вы знаете, как работает функция и какой результат она даст в определенных ситуациях. Поэтому вы просто проверяете, соответствует ли ожидаемый результат фактическому результату функции. Это то, что мы делали на *этапе 3*. Мы знаем, что функция `f1.is_not_null()` возвращает значение `true`, и проверили это. На *этапе 4* мы знаем, что `f1` не равно `f2`, поэтому это мы тоже проверили. Вызов `f1.throws()` создает исключение `std::logic_error`, и мы проверяем, что выбрасывается исключение ожидаемого типа.

На *этапе 2* мы создаем тестовый случай – набор проверок для оценки правильности поведения структуры foo. У нас может быть несколько тестовых случаев в одном исходном файле. Например, если мы добавим следующий код:

```
BOOST_AUTO_TEST_CASE(test_no_2) {
    foo f1{1}, f2{2};
    BOOST_REQUIRE_NE(f1, f2);
    // ...
} // BOOST_AUTO_TEST_CASE(test_no_2)
```

Этот код будет выполняться наряду с тестовым случаем test_no_1.

Параметр, передаваемый в макрос BOOST_AUTO_TEST_CASE, – это просто уникальное имя тестового случая, который отображается в случае ошибки.

```
Running 2 test cases...
main.cpp(15): error in "test_no_1": check f1.is_not_null() failed
main.cpp(17): error in "test_no_1": check f1 != f2 failed [0 == 0]
main.cpp(19): error in "test_no_1": exception std::logic_error is expected
main.cpp(24): fatal error in "test_no_2": critical check f1 != f2 failed [0 == 0]

*** 4 failures detected in test suite "test_module_name"
```

Между макросами BOOST_REQUIRE_* и BOOST_CHECK_* есть небольшая разница. Если проверка макроса BOOST_REQUIRE_* завершается неудачно, выполнение текущего теста останавливается, и Boost.Test запускает следующий тест. Однако в случае сбоя BOOST_CHECK_* выполнение текущего тестового случая не останавливается.

Этап 1 требует дополнительной осторожности. Обратите внимание на определение макроса BOOST_TEST_MODULE. Этот макрос должен быть определен до подключения заголовочных файлов Boost.Test; в противном случае ваша программа не слинкуется. Более подробную информацию о сборке можно найти в разделе «*См. также*» этого рецепта.

Дополнительно...

Некоторые читатели могут задаться вопросом: почему на этапе 4 вместо BOOST_CHECK (f1! = F2) мы написали BOOST_CHECK_NE (f1, f2)? Ответ прост: в случае провала проверки макрос на этапе 4 зачастую обеспечивает более читабельный и подробный вывод сообщения об ошибке.

В стандарте C++ отсутствует поддержка модульного тестирования. Однако библиотеку Boost.Test можно использовать для тестирования современного кода на C++ и для тестирования кода, использующего стандарты, предшествовавшие C++11.

Помните, что чем больше у вас тестов, тем более надежный код вы получаете!

См. также

- Рецепт «*Объединение нескольких тестовых случаев в одном тестовом модуле*» содержит больше информации о тестировании и макросе BOOST_TEST_MODULE;

- обратитесь к официальной документации Boost по адресу http://boost.org/libs/test для получения полного списка тестовых макросов и информации о расширенных возможностях `Boost.Test`.

Объединение нескольких тестовых случаев в одном тестовом модуле

Написание автоматических тестов полезно для вашего проекта. Однако сложно управлять тестовыми случаями, когда проект большой и над ним работает много разработчиков. В этом рецепте мы рассмотрим, как запускать отдельные тесты и как объединять несколько тестовых случаев в одном модуле.

Давайте представим, что два разработчика тестируют структуру `foo`, объявленную в заголовочном файле `foo.hpp`, и мы хотим предоставить им отдельные исходные файлы для написания тестов. В этом случае оба разработчика не будут мешать друг другу и могут работать параллельно. Однако тестовый прогон по умолчанию должен выполнять тесты обоих разработчиков.

Подготовка

Для этого рецепта требуются базовые знания C++. Этот рецепт частично использует код из предыдущего рецепта. Во время сборки вам необходимо определить макрос `BOOST_TEST_DYN_LINK` и линковаться с библиотеками `boost_unit_test_framework` и `boost_system`.

Как это делается...

Этот рецепт использует код из предыдущего рецепта. Это очень полезный рецепт для тестирования больших проектов. Не стоит недооценивать его.

1. В `main.cpp` из предыдущего рецепта оставьте только эти две строки:

```
#define BOOST_TEST_MODULE test_module_name
#include <boost/test/unit_test.hpp>
```

2. Давайте переместим тестовые случаи из предыдущего примера в два разных исходных файла:

```
// developer1.cpp
#include <boost/test/unit_test.hpp>
#include "foo.hpp"
BOOST_AUTO_TEST_CASE(test_no_1) {
    // ...
}
//////////////////////////////////////////////////////////
// developer2.cpp
#include <boost/test/unit_test.hpp>
#include "foo.hpp"
BOOST_AUTO_TEST_CASE(test_no_2) {
    // ...
}
```

Готово! Таким образом, после компиляции и линковки всех исходных файлов при выполнении программы будут тесты из обоих файлов.

Как это работает...

Вся магия выполняется макросом `BOOST_TEST_MODULE`. Если он определен перед заголовком `<boost/test/unit_test.hpp>`, то `Boost.Test` считает, что этот исходный файл является основным и в него должна быть помещена вся вспомогательная инфраструктура для тестирования. В противном случае `<boost/test/unit_test.hpp>` предоставляет только макросы для написания тестов.

Все тесты `BOOST_AUTO_TEST_CASE` будут выполняться, если вы прилинкуете их к исходному файлу, который содержит макрос `BOOST_TEST_MODULE`. При работе над большим проектом каждый разработчик может активировать компиляцию и линковку только своих собственных исходных файлов. Это позволяет быть независимым от других разработчиков и увеличивает скорость разработки – не нужно компилировать чужие файлы и запускать чужие тесты при отладке.

Дополнительно...

Библиотека `Boost.Test` может выборочно запускать тесты. Мы можем выбрать тесты для запуска и передать их в виде аргументов командной строки. Например, эта команда запускает только тестовый случай `test_no_1`:

```
./testing_advanced -run=test_no_1
```

Следующая команда запускает два тестовых случая:

```
./testing_advanced -run=test_no_1,test_no_2
```

К сожалению, стандарт C++ не имеет встроенной поддержки тестирования, и похоже, что C++23 также не будет использовать аналоги классов и методов `Boost.Test`.

См. также

- Рецепт «*Написание тестовых случаев*» содержит дополнительную информацию о библиотеке `Boost.Test`. Прочтите официальную документацию по адресу http://boost.org/libs/test для получения дополнительной информации о `Boost.Test`;
- смельчаки могут попытаться и посмотреть исходные коды тестов библиотек Boost. Они находятся в подпапке `libs`, расположенной в папке `boost`. Например, тестовые случаи `Boost.LexicalCast` размещены в `boost_1_XX_0/libs/lexical_cast/test`.

Манипулирование изображениями

На десерт я оставил вам кое-что очень вкусное – Generic Image Library, или просто `Boost.GIL`, которая позволяет манипулировать изображениями, не слишком заботясь об их формате.

Давайте сделаем что-нибудь простое и интересное, используя эту библиотеку. Например, создадим программу, которая превращает любое изображение в негатив.

Подготовка

Этот рецепт требует базовых знаний C++, шаблонов и `Boost.Variant`. При сборке линкуйтесь с библиотекой `png`.

Как это делается...

Чтобы было проще, мы будем работать только с изображениями в формате PNG.

1. Начнем с подключения заголовочных файлов:

```
#include <boost/version.hpp>
#if (BOOST_VERSION < 106800)
#   include <boost/gil/gil_all.hpp>
#   include <boost/gil/extension/io/png_dynamic_io.hpp>
#else
#   include <boost/gil.hpp>
#   include <boost/gil/extension/io/png/old.hpp>
#endif
#include <boost/mpl/vector.hpp>
```

2. Теперь нам нужно определить типы изображений, с которыми мы хотим работать:

```
int main(nt argc, char *argv[]) {
    typedef boost::mpl::vector<
            boost::gil::gray8_image_t,
            boost::gil::gray16_image_t,
            boost::gil::rgb8_image_t
    > img_types;
```

3. Открытие существующего изображения в формате PNG можно реализовать так:

```
std::string file_name(argv[1]);
boost::gil::any_image<img_types> source;
boost::gil::png_read_image(file_name, source);
```

4. Нужно применить операцию к изображению следующим образом:

```
boost::gil::apply_operation(
    view(source),
    negate()
);
```

5. Эта строка кода поможет вам записать изображение в файл:

```
boost::gil::png_write_view("negate_" + file_name, const_view(source));
```

6. Давайте посмотрим на структуру для преобразования изображения:

```
struct negate {
    typedef void result_type; // необходимый для работы псевдоним типа

    template <class View>
    void operator()(const View& source) const {
```

```
            // ...
        }
    }; // negate
```

7. Тело `operator()` состоит из получения типа канала:

```
typedef typename View::value_type value_type;
typedef typename boost::gil::channel_type<value_type>::type channel_t;
```

8. Итерируемся по пикселям и каналам:

```
const std::size_t channels = boost::gil::num_channels<View>::value;
const channel_t max_val = (std::numeric_limits<channel_t>::max)();

for (unsigned int y = 0; y < source.height(); ++y) {
     for (unsigned int x = 0; x < source.width(); ++x) {
          for (unsigned int c = 0; c < channels; ++c) {
              source(x, y)[c] = max_val - source(x, y)[c];
          }
     }
}
```

Теперь давайте посмотрим на результаты нашей программы:

Предыдущая картинка – это негатив изображения, которое идет далее:

Как это работает...

На *этапе 2* мы описываем типы изображений, с которыми хотим работать. Это серые изображения с 8 или 16 битами на пиксель и изображения RGB с 8 битами на пиксель.

Класс `boost::gil::any_image<img_types>` – это разновидность `Boost.Variant`, который может содержать изображение одного из типов, перечисленных в `img_types`. Как вы, возможно, уже догадались, `boost::gil::png_read_image` считывает изображения в переменную `source`.

Функция `boost::gil::apply_operation` на этапе 4 делает почти то же самое, что и `boost::apply_visitor` из библиотеки `Boost.Variant`, – применяет операцию к содержимому. Обратите внимание на использование `view(source)`. Функция `boost::gil::view` создает легкую обертку вокруг изображения, которая интерпретирует его как двумерный массив пикселей.

Вы помните, что для `Boost.Variant` мы выводили тип результата с помощью наследования от `boost::static_visitor`? Когда мы используем GIL, вместо наследования нам нужно завести псевдоним типа с именем `result_type` внутри посетителя. Это можно увидеть *на этапе 6*.

Немного теории: изображения состоят из точек, которые называют **пикселями**. У изображения все пиксели одного типа. Однако пиксели разных изображений могут различаться по количеству каналов и количеству битов для передачи цвета в рамках одного канала. Канал представляет один цвет. В случае RGB-изображения у нас есть пиксель, состоящий из трех каналов – красного, зеленого и синего. В случае серого изображения у нас пиксель содержит один канал, представляющий серый цвет.

Вернемся к нашему изображению. На *этапе 2* мы описали типы изображений, с которыми хотим работать. На *этапе 3* один из этих типов изображений считывается из файла и сохраняется в переменной `source`. На *этапе 4* метод `operator()` структуры `negate` инстанцируется для всех типов изображений.

На *этапе 7* мы видим, как получить тип канала из изображения.

На *этапе 8* мы перебираем пиксели и каналы, превращаем пиксели в негатив. Процесс превращения изображения в негатив осуществляется с помощью выражения `max_val - source(x, y)[c]`, а результат записывается обратно в изображение.

На *этапе 5* мы записываем изображение в файл.

Дополнительно...

В стандарте C++ нет встроенных методов для работы с изображениями и в ближайшее десятилетие вряд ли появится. Продолжается работа по добавлению возможностей для рисования 2D-графики в стандартную библиотеку C++, хотя это своего рода ортогональная функциональность.

Библиотека `Boost.GIL` работает быстро и эффективно. Компиляторы неплохо оптимизируют ее код, и мы можем даже помочь оптимизатору, используя некоторые методы из `Boost.GIL` для развертывания циклов. Но в этой главе рассказывается только об основах библиотеки, поэтому пора остановиться.

См. также

- Более подробную информацию о `Boost.GIL` можно найти в официальной документации на странице http://boost.org/libs/gil;
- *см.* рецепт «*Хранение одного из нескольких выбранных типов в переменной или контейнере*» главы 1 «*Приступаем к написанию приложения*» для получения дополнительной информации о библиотеке `Boost.Variant`;
- посетите сайт https://isocpp.org/ для получения дополнительных новостей о C++;
- посетите сайт https://stdcpp.ru/, где обсуждаются предложения к стандарту C++ на русском языке.

Предметный указатель

Атомарные операции 147
Взаимоблокировка типа ABBA 164
Дерево квадрантов.
См. Квадродерево
Исключение 34
Компилятор 57, 68, 147, 241
Массив 221
Общая память 311, 314
Переменные 37, 162
Потоки 137
Сопрограммы 299, 323
Типы 272
Указатели 315

B

bimap 266, 268, 269
Boost 29, 30, 31, 33, 34, 35, 36, 37, 38, 40, 41, 42, 44, 46, 47, 48, 49, 51, 52, 53, 54, 56, 58, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100, 101, 103, 104, 105, 106, 108, 109, 110, 112, 114, 115, 116, 117, 119, 120, 122, 123, 125, 127, 128, 129, 130, 131, 132, 133, 134, 135, 137, 139, 140, 141, 146, 147, 148, 152, 155, 157, 158, 159, 160, 161, 163, 164, 166, 167, 168, 171, 172, 174, 175, 176, 180, 182, 183, 184, 187, 188, 190, 194, 196, 198, 200, 201, 202, 204, 205, 206, 207, 210, 211, 213, 215, 216, 218, 220, 221, 222, 223, 224, 225, 226, 228, 229, 230, 233, 234, 236, 237, 238, 239, 242, 244, 245, 246, 247, 248, 249, 250, 251, 252, 253, 254, 256, 257, 258, 261, 262, 263, 264, 265, 266, 268, 269, 270, 272, 274, 276, 277, 278, 280, 281, 283, 284, 285, 286, 287, 289, 291, 292, 293, 294, 296, 297, 298, 299, 301, 302, 303, 304, 305, 306, 307, 308, 309, 310, 311, 314, 315, 317, 319, 320, 322, 323, 325, 326, 328, 329, 331, 332, 333, 334, 335, 336, 337, 338, 339, 340, 342, 343
Boost.Algorithm reference 63, 64
Boost.Asio library reference 171, 172, 174, 175, 176, 180, 182, 183, 184, 187, 188, 190, 198, 204
Boost.Core library reference 58, 120, 123, 125
Boost.Function library reference 78, 79, 80, 81, 82, 198, 202, 262, 319, 323
Boost.Graph library reference 325, 326, 329, 331, 332
Boost.Interprocess 309, 310, 311, 314, 315, 317, 319
Boost.MPL library reference 127, 128, 129, 131, 132, 133, 226, 228, 229, 230, 233, 234, 237, 238, 239, 242, 246, 247, 248, 249, 252
Boost.MultiIndex reference 272, 274, 275
Boost.Optional library reference 42, 44, 45
Boost.Thread documentation reference 69, 72, 75, 137, 139, 140, 141, 152, 155, 157, 158, 159, 161, 164, 166, 168, 183, 190, 196, 298, 319, 323
Boost.Variant library reference 36, 37, 38, 40, 41, 42, 225, 236, 262, 340, 342, 343

C

C++11 45, 47, 60, 290

Coroutines TS 323

G

Graphviz 329, 330, 331, 332
Gvedit 331

I

if constexpr 132, 238

Q

Q-матрица. См. Матрица состояний

W

work_queue 137, 149, 150, 151, 152, 155, 168, 191, 192, 193, 194, 198, 312, 313, 314

Книги издательства «ДМК Пресс» можно заказать
в торгово-издательском холдинге «Планета Альянс» наложенным платежом,
выслав открытку или письмо по почтовому адресу:
115487, г. Москва, 2-й Нагатинский пр-д, д. 6А.
При оформлении заказа следует указать адрес (полностью),
по которому должны быть высланы книги;
фамилию, имя и отчество получателя.
Желательно также указать свой телефон и электронный адрес.
Эти книги вы можете заказать и в интернет-магазине: **www.a-planeta.ru**.
Оптовые закупки: тел. **(499) 782-38-89**.
Электронный адрес: **books@alians-kniga.ru**.

Антон Полухин

Распределенные системы реального времени

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Редактор *Полухин А. А.*
Перевод *Беликов Д. А.*
Корректор *Синяева Г. И.*
Верстка *Луценко С. В.*
Дизайн обложки *Мовчан А. Г.*

Формат 70×100 1/16.
Гарнитура «PT Serif». Печать цифровая.
Усл. печ. л. 25,3. Тираж 200 экз.

Веб-сайт издательства: www.dmkpress.com

Отпечатано в ООО «Принт-М»
142300, Московская обл., Чехов, ул. Полиграфистов, 1