



Райнер Гримм

Стандартная библиотека С++ в примерах и пояснениях

The C++ Standard Library

What every professional C++ programmer
should know about the C++ standard library

Rainer Grimm



www.ModernesCpp.de

Стандартная библиотека C++ в примерах и пояснениях

Все, что должен знать каждый профессиональный
программист на C++ о стандартной библиотеке

Райнер Гримм

УДК 004.438C++
ББК 32.973.2
Г84

Г84 Райнер Гримм

Стандартная библиотека C++ в примерах и пояснениях / пер. с англ.
А. В. Логунова. – М.: ДМК Пресс, 2025. – 336 с.: ил.

ISBN 978-5-93700-380-5

Книга представляет собой компактный справочник по стандартной библиотеке языка программирования C++, обновленной до версии стандарта C++23. В ней изложена вся необходимая информация, которую должен знать о стандартной библиотеке профессиональный программист на C++. Вы познакомитесь с различными библиотеками в рамках общей стандартной библиотеки и стандартной библиотеки шаблонов C++ и научитесь пользоваться мощным арсеналом вспомогательных инструментов.

Книга предназначена читателям, знакомым с программированием на C++ и желающим иметь под рукой удобное руководство по работе с новой версией стандарта.

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.

Материал, изложенный в данной книге, многократно проверен. Но, поскольку вероятность технических ошибок все равно существует, издательство не может гарантировать абсолютную точность и правильность приводимых сведений. В связи с этим издательство не несет ответственности за возможные ошибки, связанные с использованием книги.

ISBN 979-8-38665-859-5 (англ.)
ISBN 978-5-93700-380-5 (рус.)

Copyright © 2023 Rainer Grimm
© Оформление, перевод на русский язык,
издание, ДМК Пресс, 2025

Оглавление

От издательства.....	12
Об авторе.....	13
Введение	14
Назначение этой книги.....	14
Условные обозначения	14
Примеры исходного кода.....	15
Исходный код	15
Значение и объект.....	15
Признательности.....	15
Дальнейшая информация.....	16
Циппи.....	16
1. Стандартная библиотека	17
Краткий обзор.....	17
История	17
Краткий обзор.....	18
Вспомогательные инструменты.....	19
Стандартная библиотека шаблонов.....	19
Библиотеки для работы с числами.....	22
Библиотеки для работы с текстом.....	22
Библиотека для работы с вводом и выводом данных	23
Библиотеки для работы с многопоточностью.....	23
Использование библиотек	25
Вставка заголовочных файлов.....	25
Импорт стандартной библиотеки	25
Использование именных пространств	26
Сборка исполняемого файла	28
2. Вспомогательные инструменты	29
Краткий обзор.....	29
Вспомогательные функции.....	29
std::min, std::max и std::minmax	30
std::midpoint и std::lerp	31
std::move	33
std::forward.....	33
std::to_underlying.....	35
std::swap	35
Адаптеры функциональных единиц кода	35
std::bind.....	35
std::bind_front (C++20)	37
std::bind_back (C++23).....	37

std::function	37
Пары	38
std::make_pair.....	39
Кортежи	39
std::make_tuple	40
std::tie и std::ignore	40
Обертки вокруг ссылок.....	41
std::ref и std::cref	42
Умные указатели	43
std::unique_ptr.....	44
Специальные удалители	47
std::make_unique	47
std::shared_ptr	47
std::make_shared	48
std::shared_ptr из указателя this	49
std::weak_ptr	49
Циклические ссылки	51
Признаки типов	52
Проверка информации о типе.....	53
Отношения типов.....	56
Модификации типов	57
Операции на признаках.....	58
Отношения между членами	59
Константно-вычисляемый контекст.....	59
Библиотека для работы со временем	60
Временная точка	61
Продолжительность времени	62
Часы	64
Время дня.....	65
Календарь	65
Часовой пояс.....	66
Функциональность библиотеки chrono по вводу-выводу	68
Новые обобщенные типы std::any, std::optional и std::variant.....	68
3. Интерфейс всех контейнеров.....	78
Краткий обзор.....	78
Создание и удаление	79
Размер контейнеров.....	80
Доступ к элементам контейнера	81
Присваивание и обмен контейнеров местами.....	83
Сравнение контейнеров.....	84
Стирание контейнеров.....	85
4. Контейнеры последовательностей	87
Краткий обзор.....	87
Массивы.....	89

Векторы	90
Размер в сопоставлении с емкостью	91
Очередь с двусторонним доступом	93
Списки	94
Впереднаправленные списки	96
5. Ассоциативные контейнеры	99
Краткий обзор	99
Проверка на наличие элемента	101
Вставка и удаление	102
Упорядоченные ассоциативные контейнеры	103
Краткий обзор	103
Ключи и значения	103
Критерий сравнения	104
Специальные функции поиска	105
Неупорядоченные ассоциативные контейнеры	107
Краткий обзор	107
Ключи и значения	108
Производительность	108
Хеш-функция	109
6. Адаптеры контейнеров	112
Краткий обзор	112
Линейные контейнеры	112
Стек	113
Очередь	114
Очередь с приоритетом	115
Ассоциативные контейнеры	116
std::sorted_unique	117
7. Виды на последовательности объектов	118
Краткий обзор	118
Доступ к смежно расположенным объектам	118
Многомерный доступ	122
8. Итераторы	128
Краткий обзор	128
Категории итераторов	129
Критерий итератора	130
Полезные функции	131
Адаптеры итераторов	133
Итераторные адаптеры вставки	133
Потоковые итераторные адаптеры	134
9. Вызываемые единицы кода	136
Краткий обзор	136
Функции	137

Функциональные объекты	137
Предопределенные функциональные объекты	138
Лямбда-функции	139
10. Стандартные алгоритмы	140
Краткий обзор.....	140
Общепринятые правила.....	141
Итераторы как связующее звено.....	143
Последовательное, параллельное либо параллельно-векторизованное исполнение	143
Политики исполнения	143
Параллельные версии стандартных алгоритмов	145
for_each.....	146
Немодифицирующие стандартные алгоритмы	148
Поиск элементов	148
Подсчет элементов	149
Проверка условий на диапазонах	150
Сравнение диапазонов	151
Поиск диапазонов внутри диапазонов.....	153
Модифицирующие стандартные алгоритмы	155
Копирование элементов и диапазонов	155
Замена элементов и диапазонов.....	156
Удаление элементов и диапазонов	157
Заполнение и создание диапазонов	159
Перемещение диапазонов.....	160
Обмен диапазонами.....	161
Преобразование диапазонов.....	161
Инверсия диапазонов	162
Поворот диапазонов	163
Сдвиг диапазонов.....	164
Произвольная перетасовка диапазонов	165
Удаление дубликатов	166
Разбиение диапазонов	167
Сортировка	169
Двоичный поиск.....	171
Операции слияния.....	172
Кучи	176
Минимум и максимум.....	178
Перестановки.....	180
Численные алгоритмы	181
Новые параллельные алгоритмы в стандарте C++17.....	183
Неинициализированная память	187
11. Диапазоны.....	189
Краткий обзор.....	189
Диапазон	190

std::ranges	190
Сторожок.....	191
Вид	192
Адаптеры диапазонов	192
std::generator	195
Работа прямо на контейнерах	195
Композиция функций	197
Ленивое оценивание	198
Алгоритмы из std в сопоставлении с алгоритмами из std::ranges.....	199
12. Числа	201
Краткий обзор.....	201
Случайные числа	201
Генератор случайных чисел.....	201
Распределение случайных чисел.....	202
Числовые функции, унаследованные у языка C	204
Математические константы	206
13. Строковые объекты	208
Краткий обзор.....	208
Создание и удаление	210
Конвертация между последовательностями символов C++ и C	211
Размер в сопоставлении с емкостью	212
Сравнение	214
Конкатенация строковых объектов.....	214
Доступ к элементам.....	215
Ввод и вывод данных	216
Поиск	217
Проверка на наличие подпоследовательности символов	220
Проверка на наличие префикса или суффикса.....	220
Проверка на наличие содержащейся подпоследовательности символов.....	220
Модифицирующие операции	221
Числовые конвертации	223
14. Виды на строковые объекты	226
Краткий обзор.....	226
Создание и инициализация.....	227
Немодифицирующие операции	228
Модифицирующие операции	228
15. Регулярные выражения	231
Краткий обзор.....	231
Типы символов	232
Объекты регулярного выражения	233
Результат поиска match_results.....	234
std::sub_match.....	236

Совпадение	237
Поиск	238
Замена	239
Форматирование	241
Повторный поиск	242
std::regex_iterator	242
std::regex_token_iterator	243
16. Потоки ввода и вывода	245
Краткий обзор	245
Иерархия	246
Функции ввода и вывода	247
Извлечение данных (ввод)	248
Форматированное извлечение	248
Неформатированное извлечение	249
Вставка данных (вывод)	250
Конкретизатор формата	250
Потоки данных	253
Строковые потоки	253
Файловые потоки	255
Произвольный доступ	257
Состояние потока данных	258
Пользовательские типы данных	260
17. Форматирование	262
Краткий обзор	262
Форматирующие функции	262
std::vformat, std::vformat_to и std::make_format_args	263
std::print и std::println	263
Синтаксис	264
Конкретизация формата	265
Символы заполнения и выравнивание текста	265
Знак, ширина и прецизионность чисел	265
Типы данных	266
Пользовательский форматизатор	266
Форматирование контейнера std::vector	267
18. Файловая система	269
Краткий обзор	269
Классы	271
Манипулирование разрешениями на доступ к файлу	272
Функции-члены	273
Чтение и установка времени последней записи файла	274
Информация о пространстве в файловой системе	275
Типы файлов	276
Получение типа файла	277

19. Многопоточность	279
Краткий обзор.....	279
Модель памяти.....	279
Атомарные типы данных	280
std::atomic_flag	280
std::atomic.....	281
Фундаментальный интерфейс атомарных типов.....	281
Пользовательские атомарные типы std::atomic<user-defined type>	282
Все атомарные операции.....	285
Потоки инструкций	287
std::thread	287
Создание	287
Время жизни.....	288
Аргументы	289
Операции	291
Автоматическое присоединение	292
Сигнал остановки	293
std::stop_token, std::stop_source и std::stop_callback	294
Коллективные переменные	296
Гонка за данными.....	296
Мьютексы.....	298
Взаимная бокировка	300
Блокировочные замки	302
Потокобезопасная инициализация	305
Потоково-локальные данные	306
Условные переменные	307
Семафоры.....	310
Координационные типы	312
std::latch.....	312
std::barrier.....	314
Задания.....	315
Потоки инструкций в сопоставлении с заданиями	315
Синхронизация	321
20. Сопрограммы.....	323
Краткий обзор.....	323
co_yield	324
co_await	324
Типы, допускающие ожидание.....	325
Бесконечный поток данных посредством co_yield.....	325
Предметный указатель.....	329

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 на нашем веб-сайте www.dmkpress.com, зайдя на страницу книги и оставив комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу dmkpress@gmail.com; при этом укажите название книги в теме письма.

Если вы являетесь экспертом в какой-либо области и заинтересованы в написании новой книги, заполните форму на нашем веб-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 обеспечить высокое качество наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг, мы будем очень благодарны, если вы сообщите о ней главному редактору по адресу dmkpress@gmail.com. Сделав это, вы избавите других читателей от недопонимания и поможете нам улучшить последующие издания этой книги.

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 незаконной публикацией какой-либо из наших книг, пожалуйста, пришлите нам ссылку на интернет-ресурс, чтобы мы могли применить санкции.

Ссылку на подозрительные материалы можно прислать по адресу электронной почты dmkpress@gmail.com.

Мы высоко ценим любую помощь по защите наших авторов, благодаря которой мы можем предоставлять вам качественные материалы.

Об авторе



Райнер Гримм с 1999 года работает архитектором программного обеспечения, руководителем группы и преподавателем. В 2002 году он организовывал встречи стажеров компании по повышению квалификации. С 2002 года проводит обучающие курсы. Его первые учебные курсы были посвящены проприетарному управленческому программному обеспечению, но вскоре он начал преподавать языки программирования Python и C++. В свободное время любит писать статьи о языках C++, Python и Haskell. Райнеру также нравится выступать на конференциях.

Еженедельно публикует посты в своем английском блоге Modernes Cpp¹, а также в немецком блоге², размещенном на Heise Developer.

С 2016 года является независимым преподавателем и проводит семинары по современным языкам C++ и Python. Опубликовал несколько книг на разных языках о современном C++ и, в частности, о конкурентности. В силу своей профессии всегда находится в поиске наилучшей методики преподавания современного языка C++.

¹ См. <https://www.modernescpp.com/>.

² См. <https://www.grimm-jaud.de/index.php/blog>.

Введение

Назначение этой книги

Книга «Стандартная библиотека C++ в примерах и пояснениях» представляет собой сжатый справочник по стандартной библиотеке текущего стандарта C++23 языка программирования C++, формально именуемого «Международным стандартом ISO/IEC 14882:2024(E) – Язык программирования C++»¹. Стандарт C++23 содержит свыше 2100 страниц и подчиняется более крупному стандарту C++20. По сравнению с ним, стандарты C++23 и C++17 не отличаются особой величиной или малостью, а стандарт C++14 является небольшим дополнением к стандарту C++11. Стандарт C++11 содержит более 1300 страниц и был опубликован в 2011 году. Это произошло через 13 лет после выхода первого и единственного стандарта языка C++, C++98. Разумеется, еще есть стандарт C++03, который был опубликован в 2003 году. Но C++03 считается исправленным релизом.

Цель этого краткого справочника состоит в том, чтобы предоставить сжатое руководство по стандартной библиотеке C++. В книге я исхожу из того, что вы знакомы с языком C++, и в этом случае вы получите максимальную пользу от данного справочника. Если же язык C++ для вас в новинку, то рекомендуется начать с учебника по ядру языка C++. После освоения такого учебника вы будете во всеоружии сделать следующий большой шаг, прочитав эту книгу. В целях облегчения вашей работы в книге приводятся многочисленные короткие фрагменты исходного кода, которые призваны связывать теорию и практику.

Условные обозначения

В книге используется всего несколько типографских условных обозначений.

Особые шрифты

курсивный шрифт

используется в случаях, когда нужно выделить что-то существенное.

моноширинный жирный шрифт

используется в исходном коде, инструкциях, ключевых словах и именах типов, переменных, функций и классов.

Особые значки

Используются для выделения уникальной информации, подсказок и предупреждений.

¹ См. <https://www.iso.org/standards.html>.

**Заголовок информации**

Текст информации.

**Заголовок подсказки**

Описание подсказки.

**Заголовок предупреждения**

Описание предупреждения.

Примеры исходного кода

Я не очень люблю использовать директивы и объявления, потому что они скрывают именное пространство библиотеки. В книге они используются таким образом, чтобы происхождение всегда можно было выводить из директивы (`using namespace std;`) или объявления `using` (`using std::cout;`). Тем не менее ввиду ограниченного объема страницы их приходилось время от времени использовать.

Во фрагментах исходного кода показаны только заголовочные файлы избранной функциональности. Для булевых величин показывается `true` либо `false`, а манипулятор ввода-вывода `std::boolalpha` не используется. Если ваш компилятор поддерживает модульную организацию стандартной библиотеки по стандарту C++23, то заголовки можно заменить инструкцией `import std`.

Исходный код

Ради лаконичности в этой книге приводятся только короткие фрагменты исходного кода. Название всей программы находится в первой строке фрагмента.

Значение и объект

Экземпляры фундаментальных типов данных в книге называются значениями. Такое правило было унаследовано языком C++ у языка C. Экземпляры более сложных типов или классов, которые нередко состоят из фундаментальных типов, называются объектами. Объектами обычно являются экземпляры пользовательских типов, классов или контейнеров.

Признательности

Прежде всего хотел бы поблагодарить лектора издательства O'Reilly Александру Фоллениус за немецкую версию книги «C++ Standardbibliothek – kurz & gut»¹. Данная версия является прародителем книги, которую вы держите в руках. Неоценимую работу по корректуре книги оказали Карстен Анерт, Гунтрам

¹ См. <http://shop.oreilly.com/product/9783955619688.do>.

Берти, Дмитрий Ганюшин, Свен Йоханнсен, Торстен Робицки, Барт Вандевостейн и Феликс Винтер. Всем им огромное спасибо.

В своем англоязычном блоге www.ModernesCpp.com¹ я разместил запрос на перевод книги на английский язык и в результате получил гораздо больше откликов, чем ожидал. Особая благодарность всем вам, включая моего сына Мариуса, который вычитывал ее первым.

Вот имена в алфавитном порядке: Махеш Аттарде, Рик Ауде, Пит Барроу, Майкл Бен-Дэвид, Дэйв Бернс, Альваро Фернандес, Джульетта Гримм, Джордж Хааке, Клэр Макрэй, Арне Мерц, Ян Рив, Джейсон Тернер, Барт Вандевостин, Иван Вергилиев и Анджей Варжинский.

Дальнейшая информация

Идею книги довольно легко перефразировать следующим образом: это «все, что должен знать каждый профессиональный программист на C++ о стандартной библиотеке C++». Такой замысел книги является причиной того, что многие вопросы остались без ответа, поэтому в начале каждой новой темы даются ссылки на подробную информацию. Ссылка будет перенаправлять к отличному онлайн-ресурсу en.cppreference.com^{2,3}.

Циппи

А теперь позвольте представить вам Циппи. Циппи будет сопровождать вас по всей книге. Надеюсь, она вам понравится.



Меня зовут Циппи.
Я – любопытная, умная и,
конечно же, женственная!

¹ См. <http://www.modernescpp.com/index.php/do-you-wan-t-to-proofread-a-book>.

² См. <http://en.cppreference.com/w/>.

³ Как вариант <https://cplusplus.com/>. – Прим. перев.

1. Стандартная библиотека

Краткий обзор

Стандартная библиотека C++ представляет собой набор предопределенных типов, классов и функций, которые обеспечивают главные функциональности программ на C++, упрощают решение общих задач программирования и повышают эффективность разработки программного обеспечения. Она состоит из многочисленных компонентов, включая библиотеку ввода-вывода, библиотеку контейнеров, библиотеку стандартных алгоритмов, библиотеку по работе со строковыми объектами, библиотеку по работе с числами и библиотеку вспомогательных инструментов, где содержатся дополнительные вспомогательные средства, такие как умные указатели и функциональные объекты.

Эта глава служит двум целям. Она призвана дать краткое изложение ее функциональных возможностей и первое представление о том, как их использовать.

История

У языка программирования C++ и, соответственно, у его стандартной библиотеки долгая история. C++ появился в 1980-х годах прошлого тысячелетия и эволюционно пока что остановился на 2023 году. Любой, кто знаком с разработкой программного обеспечения, знает о той скорости, с которой развивается наша область. Так что 40 лет – это очень большой срок. Возможно, вас не очень удивит тот факт, что первые компоненты C++, такие как потоки ввода-вывода данных, были сконструированы с другим мышлением, нежели в современной стандартной библиотеке шаблонов (STL, от англ. Standard Template Library). Язык C++ начинался как объектно ориентированный, включал в свой состав обобщенное программирование посредством стандартной библиотеки шаблонов, а ныне перенял многие идеи функционального программирования. Эта эволюция развития программного обеспечения за последние 40 лет, которую можно наблюдать в стандартной библиотеке C++, также отражает и то, как решаются задачи программной реализации.

Первая стандартная библиотека стандарта C++98 была выпущена в 1998 году и состояла из трех компонентов. Это были упомянутые ранее потоки ввода-вывода данных, в основном для работы с файлами, библиотека для работы со строковыми объектами и стандартная библиотека шаблонов (STL). Стандартная библиотека шаблонов способствует прозрачному применению алгоритмов на контейнерах.



История продолжается в 2005 году стандартом ISO/IEC TR 19768 под общим названием «Технический отчет №1» (TR1). Расширение библиотеки C++ по стандарту ISO/IEC TR 19768 не было официальным, но почти все компоненты вошли в состав стандарта C++11, например библиотеки для работы с регулярными выражениями, умными указателями, хеш-таблицами, случайными числами и временем, основанные на библиотеках `boost` (<http://www.boost.org/>).

В качестве дополнения к стандартизации TR1 стандарт C++11 получил один новый компонент: библиотеку для работы с многопоточностью.

Стандарт C++14 был лишь незначительным обновлением стандарта C++11, поэтому в рамках C++14 было добавлено лишь несколько улучшений к существующим библиотекам по работе с умными указателями, кортежами, признаками типов и многопоточностью.

В стандарт C++17 вводятся библиотеки для работы с файловой системой и два новых типа данных: `std::any` и `std::optional`.

В стандарте C++20 появляются четыре выдающиеся функциональные возможности: концепты, диапазоны, сопрограммы и модули. Помимо этой большой четверки, в C++20 есть и другие жемчужины: оператор трехпутного сравнения, библиотека для работы с форматированием и типы данных, связанные с конкурентностью, – семафоры, защелки и барьеры.

В стандарте C++23 была улучшена большая четверка стандарта C++20: расширена функциональность диапазонов, появился генератор сопрограмм `std::generator`, а также модульная стандартная библиотека C++.

Краткий обзор

Поскольку в языке C++ существует большое число библиотек, зачастую довольно трудно найти библиотеку, которая удобно подходила бы для каждого варианта использования.

Вспомогательные инструменты

Вспомогательные инструменты (утилиты) – это, по сути дела, общецелевые библиотеки, которые могут применяться во многих контекстах.

Примерами вспомогательных инструментов являются функции вычисления минимума или максимума значений, срединной точки двух значений, а также перестановки значений местами или их перемещения. Благодаря безопасному сравнению целых чисел неявного приведения типов не происходит.

Другими вспомогательными инструментами являются полиморфная функциональная обертка `std::function`, вспомогательные функции `std::bind` и `std::bind_front`. С помощью функций `std::bind` или `std::bind_front` можно легко создавать новые функции из существующих. Для их связывания с переменной и последующего вызова необходима полиморфная функциональная обертка `std::function`.

С помощью вспомогательного шаблонного класса `std::pair` и его обобщения `std::tuple` можно создавать разнородные пары и кортежи произвольной длины.

На практике довольно удобны ссылочные функциональные обертки `std::ref` и `std::cref`. С их помощью можно создавать ссылочную обертку для переменной, которая для `std::cref` является константой.

Конечно же, главной изюминкой вспомогательных инструментов в языке C++ являются умные указатели. Они позволяют автоматически и в явной форме управлять памятью. Используя уникальный умный указатель `std::unique_ptr`, можно моделировать концепцию явного владения, а с помощью коллективного умного указателя `std::shared_ptr` – моделировать коллективное владение. Коллективный умный указатель `std::shared_ptr` задействует подсчет ссылок, беря на себя заботы о своем ресурсе. Третий вариант, слабый указатель `std::weak_ptr`, помогает избавляться от циклических зависимостей между коллективными указателями `std::shared_ptr`. Циклические ссылки являются классической проблемой подсчета ссылок.

Библиотека `type_traits` для работы с признаками (чертами) типов позволяет проверять, сравнивать и манипулировать информацией о типах во время компиляции.

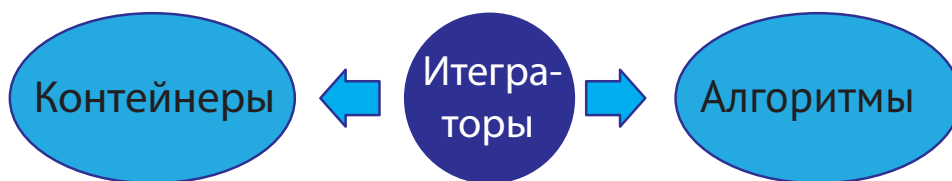
Библиотека `chrono` для работы со временем является импортируемым дополнением к новым способностям C++ в отношении многопоточности. Но она также довольно удобна для измерения производительности и включает поддержку календаря и часовых поясов.

Благодаря обобщенным типам `std::any`, `std::optional` и `std::variant` в стандарте C++17 появляются три специальных типа данных, которые могут иметь любое значение, опциональное значение либо вариант значений.

Стандартная библиотека шаблонов

Стандартная библиотека шаблонов (STL) в языке C++ – это мощная библиотека, предлагающая алгоритмы и структуры данных, которые облегчают прозрачное применение алгоритмов на контейнерах. Она предоставляет набор

распространенных алгоритмов, контейнеров, итераторов и функциональных объектов, служащих для повышения эффективности и простоты программирования на C++.



Три компонента стандартной библиотеки шаблонов

Если смотреть на стандартную библиотеку шаблонов с высоты птичьего полета, то она состоит из трех компонентов. Это контейнеры, алгоритмы и итераторы. При этом алгоритмы работают на контейнерах, а итераторы связывают их между собой. Контейнеры предъявляют лишь минимальные требования к своим элементам. Такая абстракция обобщенного программирования позволяет комбинировать алгоритмы и контейнеры уникальным образом.

В стандартной библиотеке C++ имеется богатая коллекция контейнеров, включая контейнеры последовательностей и ассоциативные контейнеры. Ассоциативные контейнеры классифицируются на упорядоченные и неупорядоченные.

Каждый контейнер последовательности имеет уникальную область применения. Тем не менее в 95 % случаев использования контейнер `std::vector` является верным выбором. Он может динамически корректировать свой размер, автоматически управляет памятью и обеспечивает отличную производительность. Напротив, контейнер `std::array` представляет собой единственный контейнер последовательностей, который не может корректировать свой размер во время исполнения. Он оптимизирован под минимальные накладные расходы на память и издержки производительности. Контейнер `std::vector` отличается тем, что помещает новые элементы в конец контейнера. С другой стороны, для размещения элемента в начале следует использовать контейнер `std::deque`. Два дополнительных контейнера – контейнер `std::list` как двусвязный список и контейнер `std::forward_list` как односвязный список – оптимизированы под высокопроизводительные операции в произвольных позициях в контейнере.

Ассоциативные контейнеры являются контейнерами пар ключ-значение. Они предоставляют свои значения по соответствующему ключу. Типичным примером использования ассоциативного контейнера является телефонный справочник, в котором ключ «фамилия» используется, чтобы извлекать значение «номер телефона». В языке C++ есть восемь ассоциативных контейнеров. С одной стороны, это ассоциативные контейнеры с упорядоченными ключами: `std::set`, `std::map`, `std::multiset` и `std::multimap`. С другой, неупорядоченные ассоциативные контейнеры: `std::unordered_set`, `std::unordered_map`, `std::unordered_multiset` и `std::unordered_multimap`.

Прежде всего, говоря об упорядоченных ассоциативных контейнерах, разница между контейнерами `std::set` и `std::map` заключается в том, что у первого нет ассоциированного значения. Разница между контейнерами `std::map`

и `std::multimap` в том, что у последнего может быть более одного одинакового ключа. Правила именования упорядоченных ассоциативных контейнеров соблюдаются и для неупорядоченных ассоциативных контейнеров, которые имеют много общего с упорядоченными. Критически важным различием между ними является производительность. Упорядоченные ассоциативные контейнеры имеют логарифмическое время доступа, а неупорядоченные ассоциативные контейнеры позволяют обращаться к элементам за постоянное время, в силу чего время доступа у неупорядоченных ассоциативных контейнеров не зависит от их размера. К контейнеру `std::unordered_map` применимо то же правило применимости, что и к контейнеру `std::vector`. Если не требуются отсортированные ключи, то в 95 % всех вариантов использования в первую очередь следует выбирать контейнер `std::unordered_map`.

Адаптеры контейнеров предоставляют упрощенный интерфейс к контейнерам последовательностей. В языке C++ есть адаптеры `std::stack`, `std::queue` и `std::priority_queue`.

Массив `C`, контейнер `std::array`, контейнер `std::vector` и контейнероподобный строковый объект класса `std::string` поддерживают виды на последовательности объектов. Вспомогательный шаблонный, или параметризуемый, класс `std::span` является видом на последовательность смежно расположенных объектов. Вид не может являться владельцем.

Связующим звеном между контейнерами и алгоритмами являются итераторы. Контейнер их создает. Итераторы можно использовать как обобщенные указатели, для того чтобы выполнять итерации по элементам в цикле в прямом и обратном направлениях или чтобы переходить к произвольной позиции в контейнере. Тип получаемого итератора зависит от контейнера. При использовании адаптера итератора можно напрямую обращаться к потоку данных.

Стандартная библиотека шаблонов предоставляет более 100 стандартных алгоритмов. Большинство алгоритмов можно выполнять последовательно, параллельно либо параллельно с векторизацией, задавая политику исполнения. Алгоритмы оперируют на элементах или диапазонах элементов. Диапазон определяется двумя итераторами. Первый, именуемый начальным итератором, определяет начало диапазона, а второй, именуемый конечным итератором, – конец диапазона. Важно помнить, что конечной итератор указывает на позицию, находящуюся сразу за последним элементом диапазона.

Эти стандартные алгоритмы можно использовать в самых разных приложениях. Можно отыскивать элементы, их подсчитывать, отыскивать диапазоны, их сравнивать или преобразовывать. Существуют алгоритмы для генерирования элементов, их замены в контейнере или удаления элементов из контейнера. Конечно же, контейнер можно сортировать, переставлять или разбивать на части, а также определять его минимальный или максимальный элемент. Многие алгоритмы можно адаптировать дальше с помощью вызываемых единиц кода, таких как функции, функциональные объекты или лямбда-функции. Вызываемые единицы кода предоставляют специальные критерии для поиска или преобразования элементов. Они значительно увеличивают мощь стандартного алгоритма.

Алгоритмы библиотеки `ranges` для работы с диапазонами являются ленивыми, могут работать непосредственно на контейнере и легко собираются в композиции. Они расширяют язык C++ идеями из функционального программирования. Более того, большинство классических алгоритмов стандартной библиотеки шаблонов имеют диапазонные аналоги, которые поддерживают проекции и предоставляют дополнительные гарантии безопасности.

Библиотеки для работы с числами

Библиотека `numeric` в языке C++ предоставляет набор функций, облегчающих выполнение числовых операций на диапазонах данных. Сюда входят алгоритмы для накопления значений, выполнения преобразований, вычисления произведений и других операций.

В языке C++ имеется ряд других библиотек для работы с числами: библиотека `random` для работы со случайными числами, математические функции `cmath`, которые язык C++ унаследовал у языка C.

Кроме того, имеется унаследованная у языка C библиотека `cstdlib`, которая содержит различные стандартные функции для работы с числами, например для конвертации строковых литералов в числа и генерации случайных чисел, и библиотека `limits`, которая применяется для получения информации о пределах различных типов чисел (например, максимальные и минимальные значения для типов `int`, `float` и других).

Библиотека `random` состоит из двух частей. В одной части находится генератор случайных чисел, а в другой – распределение сгенерированных случайных чисел. Генератор случайных чисел генерирует поток чисел между минимальным и максимальным значениями, а распределение отображает случайные числа на конкретное распределение.

Благодаря языку C в C++ есть целый ряд стандартных математических функций. Например, логарифмические, экспоненциальные и тригонометрические функции.

Язык C++ поддерживает базовые и продвинутые математические константы, такие как e , π или ϕ .

Библиотеки для работы с текстом

В языке C++ есть две мощные библиотеки для обработки текста: `string` для работы со строковыми объектами и `regex` для работы с регулярными выражениями.

Конкретный класс `std::string` обладает богатой коллекцией функций-членов, служащих для анализа и модификации текста. Поскольку у него много общего с контейнером `std::vector`, к объектам конкретного класса `std::string` можно применять алгоритмы стандартной библиотеки шаблонов. Контейнероподобный объект класса `std::string` является преемником строковых объектов C, но гораздо более простым и безопасным в использовании. Строковые объекты `std::string` C++ управляют своей памятью.

В отличие от строкового объекта класса `std::string`, копирование объекта класса `std::string_view` обходится дешево. Объект класса `std::string_view` представляет собой невладеющую ссылку на строковый объект класса `std::string`.

Регулярные выражения – это язык описания текстовых шаблонов. С помощью регулярных выражений можно выявлять наличие текстового шаблона в тексте один или несколько раз. Более того, регулярные выражения позволяют заменять найденный шаблон текстом.

Библиотека для работы с вводом и выводом данных

Библиотека `iostream` для работы с потоками ввода-вывода данных присутствует с самого начала существования языка C++ и позволяет осуществлять связь с внешним миром.

В данном конкретном случае связь означает, что оператор извлечения данных из потока (`>>`) позволяет читать форматированные либо неформатированные данные из потока данных, а оператор вставки данных в поток (`<<`) – писать их в поток вывода данных. Данные можно форматировать с помощью манипуляторов.

Классы потоков данных имеют сложную иерархию. При этом важную роль играют два класса. Во-первых, строковые потоки позволяют взаимодействовать со строковыми объектами и потоками данных. Во-вторых, файловые потоки позволяют легко читать и писать файлы. Состояние потоков данных хранится во флагах, которые можно читать и которыми можно манипулировать.

После переопределения оператора ввода и оператора вывода класс получает возможность взаимодействовать с внешним миром как фундаментальный тип данных.

Библиотека `format` для работы с форматированием представляет собой безопасную и расширяемую альтернативу семейству функций `printf` и расширяет библиотеку `iostream`.

В отличие от библиотеки `iostream` для работы с потоками ввода-вывода данных, библиотека `filesystem` для работы с файловой системой была добавлена в язык C++ начиная со стандарта C++17. В основе данной библиотеки лежат три понятия: файл, имя файла и путь. Файлы могут быть каталогами, жесткими ссылками, символическими ссылками либо обычными файлами. Пути могут быть абсолютными либо относительными.

Библиотека `filesystem` поддерживает мощный интерфейс для чтения и манипулирования файловой системой.

Библиотеки для работы с многопоточностью

В опубликованном в 2011 году стандарте C++ появилась библиотека `thread`, которая служит для обработки многопоточности – одновременного исполнения нескольких потоков инструкций. В этой библиотеке есть такие базовые строительные блоки, как атомарные переменные, потоки инструкций, блокировочные замки и переменные состояния. Это основа, на которой будущие стандарты C++ смогут выстраивать более высокие абстракции. Но стандарт

C++11 уже знает механизм заданий, которые обеспечивают более высокую абстракцию, чем указанные выше базовые строительные блоки.

На низком уровне стандарт C++11 впервые предложил модель памяти и атомарные переменные. Оба компонента являются основой для четко определенного поведения в программировании обработки многопоточности.

Библиотека `jthread` в языке C++ является частью современной поддержки многопоточности, введенной в стандарт C++20. За счет нее упрощается использование потоков инструкций, предоставляя более безопасный и удобный способ управления временем жизни потоков инструкций и их взаимодействием.

Новый поток инструкций C++ немедленно приступает к работе. Он может работать на переднем либо на заднем плане и получать данные по копии либо по ссылке. Благодаря сигналу остановки улучшенный поток `std::jthread` можно прерывать.

Доступ к коллективным переменным между потоками инструкций должен быть скоординированным. Координация может осуществляться разными способами с помощью мьютексов или блокирующих замков. Но в большинстве случаев бывает достаточной защита инициализации данных, так как они не будут мутировать в течение всего времени их жизни.

Объявление переменной как потоково-локальной обеспечивает получение потоком инструкций своей копии, вследствие чего конфликт отсутствует.

Условные переменные представляют собой классическое решение для реализации рабочих процессов отправитель–получатель. В их основу положена ключевая идея, которая заключается в том, что отправитель уведомляет получателя о завершении своей работы, чтобы тот мог к ней приступить.

Семафоры являются механизмом синхронизации, который контролирует конкурентный доступ к коллективному ресурсу. Семафор имеет счетчик, значение которого больше нуля. Приобретение семафора уменьшает счетчик, а высвобождение семафора его увеличивает. Поток инструкций может получать ресурс только тогда, когда значение счетчика больше нуля.

Аналогично семафорам, координационные типы `std::latch` (защелка) и `std::barrier` (барьер) позволяют некоторым потокам инструкций блокировать работу до тех пор, пока счетчик не станет равным нулю. В отличие от барьера `std::barrier`, защелку `std::latch` можно реиспользовать в новой итерации и корректировать ее счетчик с учетом этой новой итерации.

Задания имеют много общего с потоками инструкций. Но в отличие от потоков инструкций, которые создаются программистом в явной форме, задание будет создаваться неявно средой исполнения C++. Задания похожи на каналы данных. Данные могут быть значением, исключением либо просто уведомлением. Объект-обещание помещает данные в канал данных; объект-будущее подхватывает значение.

Сопрограммы (корутины) представляют собой функции, которые могут приостанавливать и возобновлять свое исполнение, сохраняя при этом свое состояние. Сопрограммы являются обычным способом написания событийно-

управляемых приложений¹. Событийно-управляемые приложения могут быть симуляторами, играми, серверами, пользовательскими интерфейсами или даже алгоритмами. Сопрограммы обеспечивают кооперативную многозадачность². Ключевой аспект кооперативной многозадачности заключается в том, что каждое задание занимает столько времени, сколько ему нужно.

Использование библиотек

Использование библиотеки в программе предусматривает выполнение трех шагов. Сначала нужно вставить заголовочные файлы, чтобы компилятор знал имена библиотек. Это делается с помощью инструкции `#include`. В качестве альтернативы стандартную библиотеку C++23 можно импортировать, используя инструкцию `import std`; . Поскольку имена стандартных библиотек C++ находятся в именном пространстве `std`, их можно использовать на втором шаге полностью квалифицированными, в противном случае их придется импортировать в глобальное именное пространство. Третий и последний шаг состоит в указании библиотек, чтобы компоновщик мог получить исполняемый файл. Этот третий шаг зачастую является опциональным. Следующие ниже строки объясняют описанные три шага.

Вставка заголовочных файлов

Препроцессор вставляет файл, имя которого следует за инструкцией `#include`. Чаще всего это заголовочный файл. Заголовочные файлы помещаются в угловые скобки:

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



Рекомендуется указывать все необходимые заголовочные файлы

Компилятор может самостоятельно вставлять дополнительные заголовки в заголовочные файлы, и поэтому в вашей программе могут быть все необходимые заголовки, хотя вы их и не указывали. Полагаться на эту возможность не рекомендуется. Все необходимые заголовки всегда должны быть указаны в явной форме. В противном случае обновление компилятора или перенос исходного кода может привести к ошибке компиляции.

Импорт стандартной библиотеки

В стандарте C++23 можно импортировать всю стандартную библиотеку с помощью инструкции `import`:

```
import std;
```

¹ См. https://en.wikipedia.org/wiki/Event-driven_programming.

² См. <https://de.wikipedia.org/wiki/Multitasking>.

Инструкция `import std;` импортирует все, что находится в именном пространстве `std`, начиная с заголовков C++ и заканчивая заголовками-обертками языка C, такими как `std::printf` из `<cstdio>`. Также из заголовка `<new>` импортируется глобальный оператор `::operator new` или глобальный оператор `::operator delete`.

Если требуется импортировать и аналоги глобального именного пространства, такие как `::printf` из заголовка-обертки языка C `<stdio.h>`, то следует воспользоваться инструкцией `import std.compat`.



Рекомендуется отдавать предпочтение импорту модулей, а не включению заголовков

Модули имеют много преимуществ перед заголовочными файлами. Эти преимущества относятся не только к модульности стандартной библиотеки, но и к пользовательским модулям.

Модули импортируются всего один раз, и этот процесс буквально бесплатен. Порядок импорта модулей не играет роли, а вероятность дублирования имен модулей очень мала. Модули позволяют выражать логическую структуру своего исходного кода, поскольку есть возможность явно указывать имена, которые должны экспортироваться либо не экспортироваться. Вдобавок несколько модулей можно объединять в более обширный модуль и предоставлять их заказчику в виде логического пакета.



Параллельное использование заголовочных файлов и модулей

В следующих ниже фрагментах исходного кода используются обе синтаксические формы: заголовочные файлы и модули. Рекомендуется использовать только одну из них.

Использование именных пространств

Если используются квалифицированные имена, то необходимо использовать их в точности так, как они определены. При работе с каждым именным пространством необходимо проставлять оператор доступа к области видимости `::`. Во многих библиотеках стандартной библиотеки C++ используются вложенные именные пространства.

```
#include <iostream>      // либо
#include <chrono>         // либо
...
#include std;            // или
...
std::cout << "Hello world:" << '\n';
auto timeNow= std::chrono::system_clock::now();
```

Использование неквалифицированных имен

В языке C++ имена можно использовать посредством объявления `using` и директивы `using`.

Объявление `using`

Объявление `using` добавляет имя в область видимости, в которой программист применил объявление `using`:

```
#include <iostream>      // либо
#include <chrono>        // либо
...
#import std;            // или
...
using std::cout;
using std::endl;
using std::chrono::system_clock;
...
cout << "Hello world:" << endl; // неквалифицированное имя
auto timeNow= now();           // квалифицированное имя
```

Применение объявления `using` приводит к следующим ниже последствиям:

- если одинаковое имя было объявлено в той же самой области видимости, то это приводит к неоднозначному поиску и ошибке компилятора;
- если одинаковое имя было объявлено в окружающей области видимости, то оно будет скрыто объявлением `using`.

Директива `using`

Директива `using` позволяет использовать все имена именных пространств без квалификации.

```
#include <iostream>      // либо
#include <chrono>        // либо
...
#import std;            // или
...
using namespace std;
...
cout << "Hello world:" << endl; // неквалифицированное имя
auto timeNow= chrono::system_clock::now(); // частично квалифицированное имя
```

Директива `using` не добавляет имя в текущую область видимости; она только делает имя доступным. Это означает, что:

- если одинаковое имя было объявлено в той же самой области видимости, то это приводит к неоднозначному поиску и ошибке компилятора;

- имя в локальном именном пространстве скрывает имя, объявленное в окружающем именном пространстве;
- если одинаковое имя становится видимым из разных именных пространств или если имя в именном пространстве скрывает имя в глобальной области видимости, то это приводит к неоднозначному поиску и, следовательно, к ошибке компилятора.



Директивы `using` в исходных файлах рекомендуется использовать с большой осторожностью

Директивы `using` следует использовать с большой осторожностью, так как при использовании директивы `using namespace std` все имена из `std` становятся видимыми. Это относится к именам, которые случайно скрывают имена в локальном или окружающем именном пространстве.

Не рекомендуется использовать директивы `using` в заголовочных файлах. Если включить заголовок с директивой `using namespace std`, то все имена из `std` станут видимыми.

Псевдоним именного пространства

Псевдоним именного пространства задает синоним именного пространства. Ситуации, когда для длинного имени именного пространства или вложенных именных пространств удобнее использовать псевдоним, встречаются довольно часто. Например:

```
#include <chrono>      // либо
...
#import std;           // или
...
namespace sysClock= std::chrono::system_clock;
auto nowFirst= sysClock::now();
auto nowSecond= std::chrono::system_clock::now();
```

Благодаря псевдониму именного пространства к функции `now` можно обращаться квалифицированно и с псевдонимом. Псевдоним именного пространства не должен скрывать имя.

Сборка исполняемого файла

Компоновка с библиотекой в явной форме требуется лишь в редких случаях. Состав такой инструкции зависит от платформы. Например, для того чтобы получить функциональность многопоточности при использовании компилятора `g++` либо `clang++` в качестве текущего компилятора, необходимо выполнить компоновку с библиотекой `pthread`.

```
g++ -std=c++14 thread.cpp -o thread -pthread
```

2. Вспомогательные инструменты



Циппи изучает календарь

Краткий обзор

Вспомогательные инструменты (утилиты) – это полезные средства разработки, которые можно использовать в разных контекстах. Они не привязаны к определенной области. Это утверждение относится к функциям, классам и библиотекам данной главы. Ниже будут представлены функции, которые можно применять к произвольным значениям и которые можно использовать для создания новых функций и их связывания с переменными. Вспомогательные инструменты позволяют хранить любые значения произвольного типа в парах и кортежах, а также создавать ссылки на любые значения. Умные указатели C++ являются важнейшим инструментом реализации автоматического управления памятью. Для получения информации о типе используется библиотека `type_traits`.

Вспомогательные функции

Многочисленные вариации вспомогательных шаблонных функций `min`, `max` и `minmax` можно применять к значениям и спискам инициализации. Указанным функциям необходим заголовок `<algorithm>`. С другой стороны, вспомогательные шаблонные функции `std::move`, `std::forward`, `std::to_underlying` и `std::swap` определены в заголовке `<utility>`. Их можно применять к произвольным значениям.

std::min, std::max и std::minmax

Вспомогательные шаблонные функции `std::min`¹, `std::max`² и `std::minmax`³ определены в заголовке `<algorithm>`, оперируют на значениях и списках инициализации и возвращают запрошенное значение. В случае функции `std::minmax` на выходе получается объект класса `std::pair`. Первым элементом данного объекта является минимальное значение, вторым – максимальное. По умолчанию используется оператор сравнения «меньше» (<), но можно применить свой собственный оператор сравнения. Он нуждается в двух аргументах и возвращает булеву величину. Операции и функции, которые возвращают значения `true` либо `false`, называются предикатами.

Вспомогательные шаблонные функции std::min, std::max и std::minmax

```
// minMax.cpp
...
#include <algorithm>
...
using std::cout;
...
cout << std::min(2011, 2014);           // 2011
cout << std::min({3, 1, 2011, 2014, -5}); // -5
cout << std::min(-10, -5, [](int a, int b)
    { return std::abs(a) < std::abs(b); }); // -5

auto pairInt= std::minmax(2011, 2014);
auto pairSeq= std::minmax({3, 1, 2011, 2014, -5});
auto pairAbs= std::minmax({3, 1, 2011, 2014, -5}, [](int a, int b)
    { return std::abs(a) < std::abs(b); });

cout << pairInt.first << "," << pairInt.second; // 2011,2014
cout << pairSeq.first << "," << pairSeq.second; // -5,2014
cout << pairAbs.first << "," << pairAbs.second; // 1,2014
```

В приведенной ниже таблице представлены общие сведения о вспомогательных шаблонных функциях `std::min`, `std::max` и `std::minmax`.

Таблица. Вариации вспомогательных шаблонных функций `std::min`, `std::max` и `std::minmax`

Функция	Описание
<code>min(a, b)</code>	Возвращает наименьшее значение из <code>a</code> и <code>b</code>
<code>min(a, b, comp)</code>	Возвращает наименьшее значение из <code>a</code> и <code>b</code> в соответствии с предикатом сравнения <code>comp</code>

¹ См. <http://en.cppreference.com/w/cpp/algorithm/min>.

² См. <http://en.cppreference.com/w/cpp/algorithm/max>.

³ См. <http://en.cppreference.com/w/cpp/algorithm/minmax>.

Функция	Описание
<code>min(initialiser list)</code>	Возвращает наименьшее значение списка инициализации
<code>min(initialiser list, comp)</code>	Возвращает наименьшее значение списка инициализации в соответствии с предикатом сравнения <code>comp</code>
<code>max(a, b)</code>	Возвращает наибольшее значение из <code>a</code> и <code>b</code>
<code>max(a, b, comp)</code>	Возвращает наибольшее значение из <code>a</code> и <code>b</code> в соответствии с предикатом сравнения <code>comp</code>
<code>max(initialiser list)</code>	Возвращает наибольшее значение списка инициализации
<code>max(initialiser list, comp)</code>	Возвращает наибольшее значение списка инициализации в соответствии с предикатом сравнения <code>comp</code>
<code>minmax(a, b)</code>	Возвращает наименьшее и наибольшее значения из <code>a</code> и <code>b</code>
<code>minmax(a, b, comp)</code>	Возвращает наименьшее и наибольшее значения из <code>a</code> и <code>b</code> в соответствии с предикатом сравнения <code>comp</code>
<code>minmax(initialiser list)</code>	Возвращает наименьшее и наибольшее значения списка инициализации
<code>minmax(initialiser list, comp)</code>	Возвращает наименьшее и наибольшее значения списка инициализации в соответствии с предикатом сравнения <code>comp</code>

std::midpoint и std::lerp

Вспомогательная шаблонная функция `std::midpoint(a, b)` вычисляет срединую точку между `a` и `b`, где `a` и `b` могут быть целыми числами, числами с плавающей точкой либо указателями. Если `a` и `b` – это указатели, то они должны указывать на один и тот же массивоподобный объект. Для функции `std::midpoint` требуется заголовок `<numeric>`.

Вспомогательная шаблонная функция `std::lerp(a, b, t)` вычисляет линейную интерполяцию двух чисел. Для нее требуется заголовок `<math>`. На выходе она возвращает результат вычисления $a + t(b - a)$.

Вспомогательные шаблонные функции std::midpoint и std::lerp

```
// midpointLerp.cpp

#include <cmath>
#include <numeric>

...

std::cout << "std::midpoint(10, 20): " << std::midpoint(10, 20) << '\n';
```

```
for (auto v: {0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0}) {
    std::cout << "std::lerp(10, 20, " << v << "): " << std::lerp(10, 20, v) << "\n";
}
```

```
C:\Users\rainer>midpointLerp.exe

std::midpoint(10, 20): 15

std::lerp(10, 20, 0): 10
std::lerp(10, 20, 0.1): 11
std::lerp(10, 20, 0.2): 12
std::lerp(10, 20, 0.3): 13
std::lerp(10, 20, 0.4): 14
std::lerp(10, 20, 0.5): 15
std::lerp(10, 20, 0.6): 16
std::lerp(10, 20, 0.7): 17
std::lerp(10, 20, 0.8): 18
std::lerp(10, 20, 0.9): 19
std::lerp(10, 20, 1): 20

C:\Users\rainer>
```

std::cmp_equal, std::cmp_not_equal, std::cmp_less, std::cmp_greater,
std::cmp_less_equal и std::cmp_greater_equal

Вспомогательные шаблонные функции `std::cmp_equal`, `std::cmp_not_equal`, `std::cmp_less`, `std::cmp_greater`, `std::cmp_less_equal` и `std::cmp_greater_equal` определены в заголовке `<utility>` и обеспечивают безопасное сравнение целых чисел. Безопасное сравнение означает, что отрицательное знаковое целочисленное значение меньше беззнакового целочисленного, а сравнение значений, отличных от знакового или беззнакового целочисленных значений, генерирует ошибку компиляции.



Целочисленная конвертация со встроенными целыми числами

Следующий ниже фрагмент исходного кода иллюстрирует проблему сравнения знаковых и беззнаковых целочисленных значений.

```
-1 < 0u; // true
std::cmp_greater(-1, 0u); // false
```

Значение `-1` как знаковое целое число приводится к беззнаковому типу, что и обуславливает неожиданный результат.

std::move

Вспомогательная шаблонная функция `std::move`¹ определена в заголовке `<utility>` и разрешает компилятору перемещать ресурс. Значения объекта-источника переносятся в новый объект в рамках так называемой семантики перемещения. После этого источник находится в четко определенном, но двусмысленном состоянии. Чаще всего это состояние источника является дефолтным. При использовании функции `std::move` компилятор конвертирует аргумент `arg` источника в ссылку на правостороннее значение²: `static_cast<std::remove_reference<decltype(arg)>::type&&> (arg)`. Если компилятор не может применить семантику перемещения, то он откатывает к семантике копирования:

```
#include <utility>
...
std::vector<int> myBigVec(10000000, 2011);
std::vector<int> myVec;

myVec = myBigVec;           // семантика копирования
myVec = std::move(myBigVec); // семантика перемещения
```



Перемещение обходится дешевле, чем копирование

Семантика перемещения имеет два преимущества. Во-первых, вместо дорогостоящего копирования нередко целесообразно использовать дешевое перемещение, вследствие чего нет необходимости в дополнительном выделении и высвобождении памяти. Во-вторых, некоторые объекты нельзя копировать. Например, поток инструкций или блокировочный замок.

std::forward

Вспомогательная шаблонная функция `std::forward`³ определена в заголовке `<utility>` и позволяет писать шаблонные функции, которые могут идентично пересылать свои аргументы. Типичными случаями использования функции `std::forward` являются фабричные функции или конструкторы. Фабричные функции создают объект и поэтому должны передавать свои аргументы без изменений. Конструкторы часто используют свои аргументы для инициализации своего базового класса идентичными аргументами. Таким образом, функция `std::forward` представляет собой идеальный инструмент для авторов обобщенных библиотек:

¹ См. <http://en.cppreference.com/w/cpp/utility/move>.

² Правостороннее значение (rvalue) – это временное значение, которое не имеет именованного адреса и обычно представляет собой значения, которые могут быть перемещены или скопированы. С другой стороны, левостороннее значение (lvalue) – это значение, которое имеет именованный адрес в памяти и может использоваться в качестве места назначения для присваивания. – *Прим. перев.*

³ См. <http://en.cppreference.com/w/cpp/utility/forward>.

Идеальная пересылка

```
// forward.cpp
...
#include <utility>
...
using std::initializer_list;

struct MyData{
    MyData(int, double, char){};
};

template <typename T, typename... Args>
T createT(Args&&... args){
    return T(std::forward<Args>(args)... );
}

...

int a= createT<int>();
int b= createT<int>(1);

std::string s= createT<std::string>("Only for testing.");
MyData myData2= createT<MyData>(1, 3.19, 'a');

typedef std::vector<int> IntVec;
IntVec intVec= createT<IntVec>(initializer_list<int>({1, 2, 3}));
```

Шаблонная функция `createT` должна принимать свои аргументы как универсальную ссылку¹: `Args&&... args`. Универсальная ссылка, или перенаправляющая ссылка, – это ссылка на правостороннее значение в контексте автоматического вывода типа².



Вспомогательная шаблонная функция `std::forward` позволяет использовать полностью обобщенные функции в сочетании с вариативными шаблонами

Вариативный шаблон – это мощный функциональный инструмент C++, введенный в стандарт C++11, который позволяет создавать шаблоны, способные принимать произвольное количество аргументов шаблона. Если использовать вспомогательную шаблонную функцию `std::forward` вместе с вариативными шаблонами, то можно определять полностью обобщенные шаблонные функции. Такая шаблонная функция сможет принимать произвольное количество аргументов и пересылать их без изменений.

¹ См. <https://isocpp.org/blog/2012/11/universal-references-in-c11-scott-meyers>.

² Универсальные ссылки – это ссылки, объявленные с использованием `&&`, вследствие чего они могут быть как ссылками на левостороннее значение, так и ссылками на правостороннее значение и, значит, могут связывать с чем угодно. – *Прим. перев.*

std::to_underlying

Вспомогательная шаблонная функция `std::to_underlying` стандарта C++23 конвертирует перечисление `enum` в свой базисный тип `Enum`. Она приходит на помощь в выражении `static_cast<std::underlying_type<Enum>::type>(enum)` со вспомогательной шаблонной функцией `std::underlying_type` из библиотеки `type_traits`.

std::swap

Вспомогательная шаблонная функция `std::swap`¹ определена в заголовке `<utility>`. С помощью нее можно легко менять местами два объекта. В обобщенной реализации в рамках стандартной библиотеки C++ внутренне используется вспомогательная шаблонная функция `std::move`.

Семантика перемещения с помощью вспомогательной шаблонной функции `std::swap`

```
// swap.cpp
...
#include <utility>
...
template <typename T>
inline void swap(T& a, T& b) noexcept {
    T tmp(std::move(a));
    a = std::move(b);
    b = std::move(tmp);
}
```

Адаптеры функциональных единиц кода

Вспомогательные шаблонные функции `std::bind`, `std::bind_front`, `std::bind_back` и полиморфная функциональная обертка `std::function` как адаптеры функциональных единиц кода очень хорошо сочетаются друг с другом. В отличие от функций `std::bind`, `std::bind_front` или `std::bind_back`, которые позволяют создавать новые функциональные объекты на лету, полиморфная функциональная обертка `std::function` берет эти временные функциональные объекты и связывает их с переменной.

std::bind

Вспомогательная шаблонная функция `std::bind` обладает большей силой, чем функции `std::bind_front` или `std::bind_back`, поскольку она позволяет связывать аргументы с произвольной позицией.

В отличие от функций `std::bind`, `std::bind_front` и `std::bind_back`, которые предоставляют возможность создавать новые функциональные объекты на лету,

¹ См. <http://en.cppreference.com/w/cpp/algorithm/swap>.

функциональная обертка `std::function` берет эти временные функциональные объекты и связывает их с переменной. Все указанные функции являются мощными инструментами функционального программирования и нуждаются в заголовке `<functional>`.



Функции `std::bind`, `std::bind_front`, `std::bind_back` и функциональная обертка `std::function` в основном излишни

Функция `std::bind` и функциональная обертка `std::function` (в рамках стандарта TR1¹), функции `std::bind_front` и `std::bind_back` в языке C++ в основном не нужны. Во-первых, вместо вспомогательных шаблонных функций `std::bind`, `std::bind_front`, `std::bind_back` можно использовать лямбды, а во-вторых, вместо полиморфной функциональной обертки `std::function` очень часто используется автоматическое выведение типов, применяя ключевое слово `auto`, инициирующее средство автоматического выведения типа переменной компилятором.

Создание и связывание функциональных объектов

```
// bindAndFunction.cpp
...
#include <functional>
...
// for placeholder _1 and _2
using namespace std::placeholders;

using std::bind;
using std::bind_front;
using std::bind_back;
using std::function;
...
double divMe(double a, double b){ return a/b; };

function<double(double, double)> myDiv1 = bind(divMe, _1, _2); // 200
function<double(double)> myDiv2 = bind(divMe, 2000, _1);      // 200
function<double(double)> myDiv3 = bind_front(divMe, 2000);    // 200
function<double(double)> myDiv4 = bind_back(divMe, 10);       // 200
```

Благодаря функции `std::bind`² функциональные объекты можно создавать различными способами:

- связывать аргументы с произвольной позицией;
- изменять порядок следования аргументов;

¹ См. https://en.wikipedia.org/wiki/C%2B%2B_Technical_Report_1.

² См. <http://en.cppreference.com/w/cpp/utility/functional/bind>.

- вводить местозаполнители для аргументов;
- частично вычислять функции;
- вызывать только что созданные функциональные объекты, использовать их в алгоритме стандартной библиотеки шаблонов или сохранять в полиморфной функциональной обертке `std::function`.

`std::bind_front` (C++20)

Вспомогательная шаблонная функция `std::bind_front`¹ создает вызываемую обертку из вызываемой единицы кода. Вызов `std::bind_front(func, arg ...)` связывает все аргументы `arg` с передней частью функции `func` и возвращает вызываемую обертку.

`std::bind_back` (C++23)

Вспомогательная шаблонная функция `std::bind_back`² создает вызываемую обертку из вызываемой единицы кода. Вызов `std::bind_back(func, arg ...)` связывает все аргументы `arg` с задней частью функции `func` и возвращает вызываемую обертку.

`std::function`

Полиморфная функциональная обертка `std::function`³ может хранить произвольные вызываемые единицы кода в переменных. Вызываемая единица кода может быть лямбда-функцией, функциональным объектом либо функцией. Полиморфная функциональная обертка `std::function` всегда необходима, и ее нельзя заменять на ключевое слово `auto`, когда требуется явно указывать тип вызываемой единицы кода.

Таблица диспетчеризации с использованием полиморфной функциональной обертки `std::function`

```
// dispatchTable.cpp
...
#include <functional>
...
using std::make_pair;
using std::map;

map<const char, std::function<double(double, double)>> tab;
tab.insert(make_pair('+', [](double a, double b){ return a + b; }));
tab.insert(make_pair('-', [](double a, double b){ return a - b; }));
tab.insert(make_pair('*', [](double a, double b){ return a * b; }));
```

¹ См. https://en.cppreference.com/w/cpp/utility/functional/bind_front.

² См. https://en.cppreference.com/w/cpp/utility/functional/bind_back.

³ См. <http://en.cppreference.com/w/cpp/utility/functional/function>.

```

tab.insert(make_pair('/', [](double a, double b){ return a / b; }));
std::cout << tab['+'](3.5, 4.5); // 8
std::cout << tab['-'](3.5, 4.5); // -1
std::cout << tab['*'](3.5, 4.5); // 15.75
std::cout << tab['/'](3.5, 4.5); // 0.777778

```

Параметр типа в полиморфной функциональной обертке `std::function` задает тип принимаемых ею вызываемых единиц кода.

Таблица. Возвращаемый тип и тип аргументов

Тип функции	Возвращаемый тип	Тип аргументов
<code>double(double, double)</code>	<code>double</code>	<code>double</code>
<code>int()</code>	<code>int</code>	
<code>double(int, double)</code>	<code>double</code>	<code>int, double</code>
<code>void()</code>		

Пары

С помощью вспомогательного шаблонного класса¹ `std::pair`² можно создавать пары произвольных типов. Указанный класс нуждается в заголовке `<utility>`. Он имеет дефолтный конструктор, конструктор копирования и конструктор перемещения. Объекты-пары можно менять местами с помощью вспомогательной функции `std::swap(pair1, pair2)`.

Пары широко используются в стандартной библиотеке C++. Например, вспомогательная шаблонная функция `std::minmax` возвращает свой результат в виде пары, а ассоциативные контейнеры `std::map`, `std::unordered_map`, `std::multimap` и `std::unordered_multimap` управляют своими ассоциациями ключ-значение парно.

Для получения элементов пары `p` к ним можно обращаться напрямую либо через индекс. Так, с помощью атрибута `p.first` пары или вспомогательной функции `std::get<0>(p)` можно получать первый элемент пары, а с помощью атрибута `p.second` пары или вспомогательной функции `std::get<1>(p)` – второй элемент пары.

Пары поддерживают операторы сравнения `==`, `!=`, `<`, `>`, `<=` и `>=`. При сравнении двух пар на идентичность сначала сравниваются атрибуты `pair1.first` и `pair2.first`, а затем атрибуты `pair1.second` и `pair2.second`. Аналогичная стратегия соблюдается и для других операторов сравнения.

¹ Син. шаблон класса. В языке C++ термины «шаблон класса» и «шаблонный класс» могут использоваться для обозначения одного и того же понятия, т. е. класса, который определен с использованием шаблона. Термин «шаблон класса» подчеркивает, что это шаблон, из которого могут создаваться классы с конкретными типами. Термин «шаблонный класс» акцентирует внимание на том, что класс порожден шаблоном. – *Прим. перев.*

² См. <http://en.cppreference.com/w/cpp/utility/pair>.

std::make_pair

В языке C++ есть полезная вспомогательная функция `std::make_pair`¹, служащая для генерации пар без указания их типов. Функция `std::make_pair` вычисляет их типы автоматически.

Вспомогательная функция `std::make_pair`

```
// pair.cpp
...
#include <utility>
...
using namespace std;
...
pair<const char*, double> charDoub("str", 3.14);
pair<const char*, double> charDoub2= make_pair("str", 3.14);
auto charDoub3= make_pair("str", 3.14);

cout << charDoub.first << ", " << charDoub.second; // str, 3.14
charDoub.first="Str";
get<1>(charDoub)= 4.14;
cout << charDoub.first << ", " << charDoub.second; // Str, 4.14
```

Кортежи

С помощью вспомогательного шаблонного класса `std::tuple`², которому необходим заголовок `<tuple>`, можно создавать кортежи с произвольной длиной и произвольными типами. Указанный класс является обобщением вспомогательного шаблонного класса `std::pair`. Двухэлементные кортежи и пары можно конвертировать друг в друга. Как и объект-пара `std::pair`, объект-кортеж `std::tuple` имеет дефолтный конструктор, конструктор копирования и конструктор перемещения. Кортежи можно менять местами с помощью вспомогательной функции `std::swap`.

Вспомогательная шаблонная функция `std::get` позволяет обращаться к *i*-му элементу кортежа: `std::get<i-1>(t)`. С помощью вызова `std::get<type>(t)` можно напрямую обращаться к элементу типа `type`.

Кортежи поддерживают операторы сравнения `==`, `!=`, `<`, `>`, `<=` и `>=`. При сравнении двух кортежей элементы кортежей сравниваются лексикографически. Сравнение начинается с индекса 0.

¹ См. http://en.cppreference.com/w/cpp/utility/pair/make_pair.

² См. <http://en.cppreference.com/w/cpp/utility/tuple>.

std::make_tuple

Вспомогательная функция `std::make_tuple`¹ предоставляет удобный способ создания кортежа. При этом указывать типы не требуется. Компилятор выводит их автоматически.

Вспомогательная функция std::make_tuple

```
// tuple.cpp
...
#include <tuple>
...
using std::get;

std::tuple<std::string, int, float> tup1("first", 3, 4.17f);
auto tup2= std::make_tuple("second", 4, 1.1);

std::cout << get<0>(tup1) << ", " << get<1>(tup1) << ", "
           << get<2>(tup1) << '\n'; // first, 3, 4.17
std::cout << get<0>(tup2) << ", " << get<1>(tup2) << ", "
           << get<2>(tup2) << '\n'; // second, 4, 1.1
std::cout << (tup1 < tup2) << '\n'; // true

get<0>(tup2)= "Second";

std::cout << get<0>(tup2) << ", " << get<1>(tup2) << ", "
           << get<2>(tup2) << '\n'; // Second, 4, 1.1
std::cout << (tup1 < tup2) << '\n'; // false

auto pair= std::make_pair(1, true);
std::tuple<int, bool> tup= pair;
```

std::tie и std::ignore

Вспомогательная функция `std::tie`² позволяет создавать кортежи, которые ссылаются на переменные. Элементы кортежа можно игнорировать в явной форме с помощью вспомогательной функции `std::ignore`³.

Вспомогательные функции std::tie и std::ignore

```
// tupleTie.cpp
...
#include <tuple>
```

¹ См. http://en.cppreference.com/w/cpp/utility/tuple/make_tuple.

² См. <http://en.cppreference.com/w/cpp/utility/tuple/tie>.

³ См. <http://en.cppreference.com/w/cpp/utility/tuple/ignore>.

```

...
using namespace std;

int first= 1;
int second= 2;
int third= 3;
int fourth= 4;

cout << first << " " << second << " "
    << third << " " << fourth << endl;           // 1 2 3 4

auto tup= tie(first, second, third, fourth) // привязать кортеж
    = std::make_tuple(101, 102, 103, 104); // создать кортеж
                                           // и присвоить его

cout << get<0>(tup) << " " << get<1>(tup) << " " << get<2>(tup)
    << " " << get<3>(tup) << endl;           // 101 102 103 104
cout << first << " " << second << " " << third << " "
    << fourth << endl;                       // 101 102 103 104

first= 201;
get<1>(tup)= 202;

cout << get<0>(tup) << " " << get<1>(tup) << " " << get<2>(tup)
    << " " << get<3>(tup) << endl;           // 201 202 103 104
cout << first << " " << second << " " << third << " "
    << fourth << endl;                       // 201 202 103 104

int a, b;
tie(std::ignore, a, std::ignore, b)= tup;
cout << a << " " << b << endl;               // 202 104

```

Обертки вокруг ссылок

Обертка вокруг ссылки – это обертка, конструируемая путем копирования¹ и присваиваемая путем копирования², вокруг объекта типа `type&`, которая позволяет обертывать ссылку, указывающую на объект или функцию. Она определена в заголовке `<functional>` и предлагает объект, который ведет себя как ссылка, но который допускает его копирование. В отличие от классических ссылок, объекты класса `std::reference_wrapper`³ поддерживают два дополнительных варианта использования:

- их можно использовать в контейнерах стандартной библиотеки шаблонов: `std::vector<std::reference_wrapper<int>> myIntRefVector;`

¹ См. <http://en.cppreference.com/w/cpp/concept/CopyConstructible>.

² См. <http://en.cppreference.com/w/cpp/concept/CopyAssignable>.

³ См. http://en.cppreference.com/w/cpp/utility/functional/reference_wrapper.

- экземпляры классов, в которых есть объекты класса `std::reference_wrapper`, можно копировать. То есть, в общем случае, со ссылками это делать невозможно.

Функция-член `get` позволяет обращаться к ссылке: `myInt.get()`. Обертку вокруг ссылки можно использовать для инкапсуляции и исполнения вызываемой единицы кода.

Обертки вокруг ссылок

```
// referenceWrapperCallable.cpp
...
#include <functional>
...
void foo(){
    std::cout << "Invoked" << '\n';
}

typedef void callableUnit();
std::reference_wrapper<callableUnit> refWrap(foo);

refWrap();    // Вызвана
```

std::ref и std::cref

С помощью вспомогательных шаблонных функций `std::ref`¹ и `std::cref`² можно легко создавать обертки вокруг ссылок на переменные. Функция `std::ref` создает неконстантную обертку вокруг ссылки, а функция `std::cref` – константную:

Вспомогательные функции std::ref и std::cref

```
// referenceWrapperRefCref.cpp
...
#include <functional>
...
void invokeMe(const std::string& s){
    std::cout << s << ": const " << '\n';
}

template <typename T>
void doubleMe(T t){
    t*= 2;
}
```

¹ См. <http://en.cppreference.com/w/cpp/utility/functional/ref>.

² См. <http://en.cppreference.com/w/cpp/utility/functional/cref>.

```

std::string s{"string"};

invokeMe(std::cref(s));    // string

int i= 1;
std::cout << i << '\n';    // 1

doubleMe(i);
std::cout << i << '\n';    // 15

doubleMe(std::ref(i));
std::cout << i << '\n';    // 2

```

Функция `invokeMe` может быть вызвана путем взятия константной ссылки на неконстантный строковый объект `std::string`, обернув строковый объект `std::string s` в функцию `std::cref(s)`. При обертывании переменной `i` во вспомогательную функцию `std::ref` шаблонная функция `doubleMe` вызывается со ссылкой. Следовательно, переменная `i` удваивается.

Умные указатели

В языке C++ умные указатели играют крайне важную роль, так как они позволяют реализовывать управление памятью в явной форме. Помимо упраздненного указателя `std::auto_ptr`, C++ предлагает три умных указателя. Они определены в заголовке `<memory>`.

Во-первых, уникальный умный указатель `std::unique_ptr` моделирует концепцию исключительного владения. Во-вторых, коллективный умный указатель `std::shared_ptr` моделирует концепцию коллективного владения. Наконец, еще есть слабый указатель `std::weak_ptr`. Слабый указатель `std::weak_ptr` не совсем умный, потому что у него ограниченный интерфейс. Его работа состоит в разрыве циклов коллективного указателя `std::shared_ptr`. Он моделирует концепцию временного владения.

Умные указатели управляют своим ресурсом в соответствии с идиомой RAII, вследствие чего если умный указатель выходит из области видимости, то ресурс автоматически высвобождается.



Приобретение ресурсов есть инициализация

Выражение «Приобретение ресурсов есть инициализация» (от англ. Resource Acquisition Is Initialization, аббр. RAII) обозначает популярную в языке C++ технику, при которой приобретение и высвобождение ресурсов привязано ко времени жизни объекта. Память для умного указателя выделяется в конструкторе и высвобождается в деструкторе. В языке C++ деструктор вызывается автоматически, когда объект выходит из области видимости.

Таблица. Краткий обзор умных указателей

Имя	Стандарт	Описание
<code>std::auto_ptr</code> (упразднен)	C++98	Владеет ресурсом в исключительном порядке Перемещает ресурс при копировании
<code>std::unique_ptr</code>	C++11	Владеет ресурсом в исключительном порядке Не может быть скопирован
<code>std::shared_ptr</code>	C++11	Имеет счетчик ссылок на коллективную переменную Автоматически управляет счетчиком ссылок Удаляет ресурс, если счетчик ссылок равен 0
<code>std::weak_ptr</code>	C++11	Помогает разрывать циклы коллективного указателя <code>std::shared_ptr</code> Не изменяет значения счетчика ссылок

**Указатель `std::auto_ptr` использовать не рекомендуется**

В классическом стандарте C++03 есть умный указатель `std::auto_ptr`, который заботится исключительно о времени жизни ресурса. Но у данного указателя есть концептуальная проблема. При явном либо неявном копировании указателя `std::auto_ptr` ресурс, вероятно, будет перемещен, в результате чего вместо подразумеваемой семантики копирования имеется скрытая семантика перемещения и, следовательно, нередко возникает неопределенное поведение. По этой причине в стандарте C++11 указатель `std::auto_ptr` был упразднен, вместо него рекомендуется использовать уникальный умный указатель `std::unique_ptr`. Указатель `std::unique_ptr` нельзя копировать ни явно, ни неявно. Его можно только перемещать:

`std::auto_ptr` и `std::unique_ptr`

```
#include <memory>
...
std::auto_ptr<int> ap1(new int(2011));
std::auto_ptr<int> ap2 = ap1;           // ОК

std::unique_ptr<int> up1(new int(2011));
std::unique_ptr<int> up2 = up1;         // ОШИБКА
std::unique_ptr<int> up3 = std::move(up1); // ОК
```

`std::unique_ptr`

Уникальный умный указатель `std::unique_ptr`¹, реализованный в виде вспомогательного шаблонного класса, заботится исключительно о своем ресурсе. Он автоматически высвобождает ресурс, если тот выходит из области видимости. Если семантика копирования не используется, то указатель `std::unique_ptr` можно использовать в контейнерах и алгоритмах стандартной библиотеки шаб-

¹ См. http://en.cppreference.com/w/cpp/memory/unique_ptr.

лонов. Если специальный удалитель не используется, то указатель `std::unique_ptr` обходится так же дешево и работает так же быстро, как и сырой указатель.

Ниже перечислены функции-члены вспомогательного шаблонного класса `std::unique_ptr`.

Таблица. Функции-члены вспомогательного шаблонного класса `std::unique_ptr`

Имя	Описание
<code>get</code>	Возвращает указатель на ресурс
<code>get_deleter</code>	Возвращает функцию удаления
<code>release</code>	Возвращает указатель на ресурс и высвобождает его
<code>reset</code>	Сбрасывает ресурс
<code>swap</code>	Меняет ресурсы местами

В следующем ниже фрагменте исходного кода демонстрируется применение этих функций-членов:

`std::unique_ptr`

```
// uniquePtr.cpp
...
#include <utility>
...
using namespace std;

struct MyInt{
    MyInt(int i):i_(i){}
    ~MyInt(){
        cout << "Good bye from " << i_ << endl;
    }
    int i_;
};

unique_ptr<MyInt> uniquePtr1{new MyInt(1998)};
cout << uniquePtr1.get() << endl;           // 0x15b5010

unique_ptr<MyInt> uniquePtr2{move(uniquePtr1)};
cout << uniquePtr1.get() << endl;           // 0
cout << uniquePtr2.get() << endl;           // 0x15b5010
{
    unique_ptr<MyInt> localPtr{new MyInt(2003)};
}                                           // Good bye from 2003
uniquePtr2.reset(new MyInt(2011));         // Good bye from 1998
MyInt* myInt= uniquePtr2.release();
delete myInt;                             // Good by from 2011

unique_ptr<MyInt> uniquePtr3{new MyInt(2017)};
```

```
unique_ptr<MyInt> uniquePtr4{new MyInt(2022)};
cout << uniquePtr3.get() << endl;           // 0x15b5030
cout << uniquePtr4.get() << endl;           // 0x15b5010

swap(uniquePtr3, uniquePtr4);
cout << uniquePtr3.get() << endl;           // 0x15b5010
cout << uniquePtr4.get() << endl;           // 0x15b5030
```

Вспомогательный шаблонный класс `std::unique_ptr` имеет специальный инструментарий для массивов:

`std::unique_ptr`

```
// uniquePtrArray.cpp
...
#include <memory>
...
using namespace std;

class MyStruct{
public:
    MyStruct():val(count){
        cout << (void*)this << " Hello: " << val << endl;
        MyStruct::count++;
    }
    ~MyStruct(){
        cout << (void*)this << " Good Bye: " << val << endl;
        MyStruct::count--;
    }
private:
    int val;
    static int count;
};

int MyStruct::count= 0;
...
{
    // генерирует массив myUniqueArray с тремя `MyStructs`
    unique_ptr<MyStruct[]> myUniqueArray{new MyStruct[3]};
}
// 0x1200018 Hello: 0
// 0x120001c Hello: 1
// 0x1200020 Hello: 2
// 0x1200020 GoodBye: 2
// 0x120001c GoodBye: 1
// 0x1200018 GoodBye: 0
```

Специальные удалители

Вспомогательный шаблонный класс `std::unique_ptr` может параметризоваться специальными удалителями: `std::unique_ptr<int, MyIntDeleter> up(new int(2011), myIntDeleter())`. Указатель `std::unique_ptr` по умолчанию использует удалитель ресурса.

`std::make_unique`

Вспомогательная функция `std::make_unique`¹, в отличие от своей сестринской функции `std::make_shared`, в стандарте C++11 была забыта, и поэтому она была добавлена в стандарт C++14. Функция `std::make_unique` позволяет создавать уникальный умный указатель `std::unique_ptr` за один шаг: `std::unique_ptr<int> up = std::make_unique<int>(2014)`.

`std::shared_ptr`

Коллективный умный указатель `std::shared_ptr`², реализованный в виде вспомогательного шаблонного класса, делится владением ресурсом. У него есть два дескриптора: один – для ресурса и один – для счетчика ссылок. При копировании коллективного указателя `std::shared_ptr` счетчик ссылок увеличивается на единицу. Он уменьшается на единицу, если данный указатель выходит из области видимости. Если счетчик ссылок становится равным 0 и на ресурс больше не ссылаются ни один коллективный указатель `std::shared_ptr`, то среда исполнения C++ автоматически высвобождает ресурс. Высвобождение ресурса происходит именно тогда, когда последний коллективный указатель `std::shared_ptr` выходит из области видимости. Среда исполнения C++ гарантирует атомарность операции вызова счетчика ссылок. Ввиду этих накладных расходов на администрирование коллективный указатель `std::shared_ptr` нуждается в большем времени и памяти, чем сырой указатель или уникальный умный указатель `std::unique_ptr`.

В следующей ниже таблице приводятся функции-члены вспомогательного шаблонного класса `std::shared_ptr`.

Таблица. Функции-члены вспомогательного шаблонного класса `std::shared_ptr`

Имя	Описание
<code>get</code>	Возвращает указатель на ресурс
<code>get_deleter</code>	Возвращает функцию удаления
<code>reset</code>	Сбрасывает ресурс
<code>swap</code>	Меняет ресурсы местами
<code>unique</code>	Проверяет, не является ли коллективный указатель <code>std::shared_ptr</code> исключительным владельцем ресурса
<code>use_count</code>	Возвращает значение счетчика ссылок

¹ См. http://en.cppreference.com/w/cpp/memory/unique_ptr/make_unique.

² См. http://en.cppreference.com/w/cpp/memory/shared_ptr.

std::make_shared

Вспомогательная функция `std::make_shared`¹ создает ресурс и возвращает его в качестве коллективного умного указателя `std::shared_ptr`. Вместо прямого создания коллективного умного указателя `std::shared_ptr` рекомендуется отдавать предпочтение функции `std::make_shared`, потому что данная функция работает быстрее.

Следующий ниже пример исходного кода показывает типичный вариант использования коллективного умного указателя `std::shared_ptr`.

std::shared_ptr

```
// sharedPtr.cpp
...
#include <memory>
...
class MyInt{
public:
    MyInt(int v):val(v){
        std::cout << "Hello: " << val << '\n';
    }
    ~MyInt(){
        std::cout << "Good Bye: " << val << '\n';
    }
private:
    int val;
};

auto sharPtr= std::make_shared<MyInt>(1998);    // Hello: 1998
std::cout << sharPtr.use_count() << '\n';    // 1

{
    std::shared_ptr<MyInt> locSharPtr(sharPtr);
    std::cout << locSharPtr.use_count() << '\n'; // 2
}
std::cout << sharPtr.use_count() << '\n';    // 1

std::shared_ptr<MyInt> globSharPtr= sharPtr;
std::cout << sharPtr.use_count() << '\n';    // 2

globSharPtr.reset();
std::cout << sharPtr.use_count() << '\n';    // 1
sharPtr= std::shared_ptr<MyInt>(new MyInt(2011)); // Hello:2011
                                                    // Good Bye: 1998

...
// Good Bye: 2011
```

¹ См. http://en.cppreference.com/w/cpp/memory/shared_ptr/make_shared.

В приведенном выше примере вызываемой единицей кода является функциональный объект, поэтому можно легко подсчитать количество созданных экземпляров класса. Результат находится в статической переменной `count`.

`std::shared_ptr` из указателя `this`

С помощью вспомогательного шаблонного класса `std::enable_shared_from_this`¹ можно создавать дополнительные коллективные указатели `std::shared_ptr` на себя. Пользовательский класс должен наследовать у класса `std::enable_shared_from_this` с использованием публичного наследования (`class ... public`). Указанный класс поддерживает функцию-член `shared_from_this`, чтобы возвращать коллективный умный указатель `std::shared_ptr` на указатель `this`:

`std::shared_ptr_from_this`

```
// enableShared.cpp
...
#include <memory>
...
class ShareMe: public std::enable_shared_from_this<ShareMe>{
    std::shared_ptr<ShareMe> getShared(){
        return shared_from_this();
    }
};

std::shared_ptr<ShareMe> shareMe(new ShareMe);
std::shared_ptr<ShareMe> shareMe1= shareMe->getShared();

std::cout << (void*)shareMe.get() << '\n'; // 0x152d010
std::cout << (void*)shareMe1.get() << '\n'; // 0x152d010
std::cout << shareMe.use_count() << '\n'; // 2
```

В приведенном выше примере исходного кода показано, что функции-члены `get` ссылаются на один и тот же объект.

`std::weak_ptr`

Слабый указатель `std::weak_ptr`², реализованный в виде вспомогательного шаблонного класса, не является умным указателем. Он не поддерживает прозрачный доступ к ресурсу, поскольку он лишь заимствует ресурс у коллективного умного указателя `std::shared_ptr`. Слабый указатель `std::weak_ptr` не изменяет счетчик ссылок:

¹ См. http://en.cppreference.com/w/cpp/memory/enable_shared_from_this.

² См. http://en.cppreference.com/w/cpp/memory/weak_ptr.

std::weak_ptr

```

// weakPtr.cpp
...
#include <memory>
...
auto sharedPtr= std::make_shared<int>(2011);
std::weak_ptr<int> weakPtr(sharedPtr);

std::cout << weakPtr.use_count() << '\n'; // 1
std::cout << sharedPtr.use_count() << '\n'; // 1

std::cout << weakPtr.expired() << '\n'; // false

if( std::shared_ptr<int> sharedPtr1= weakPtr.lock() ) {
    std::cout << *sharedPtr << '\n'; // 2011
}
else{
    std::cout << "Don't get it!" << '\n';
}

weakPtr.reset();

if( std::shared_ptr<int> sharedPtr1= weakPtr.lock() ) {
    std::cout << *sharedPtr << '\n';
}
else{
    std::cout << "Don't get it!" << '\n'; // Don't get it!
}

```

В приведенной ниже таблице представлен краткий обзор функций-членов вспомогательного шаблонного класса `std::weak_ptr`.

Таблица. Функции-члены вспомогательного шаблонного класса `std::weak_ptr`

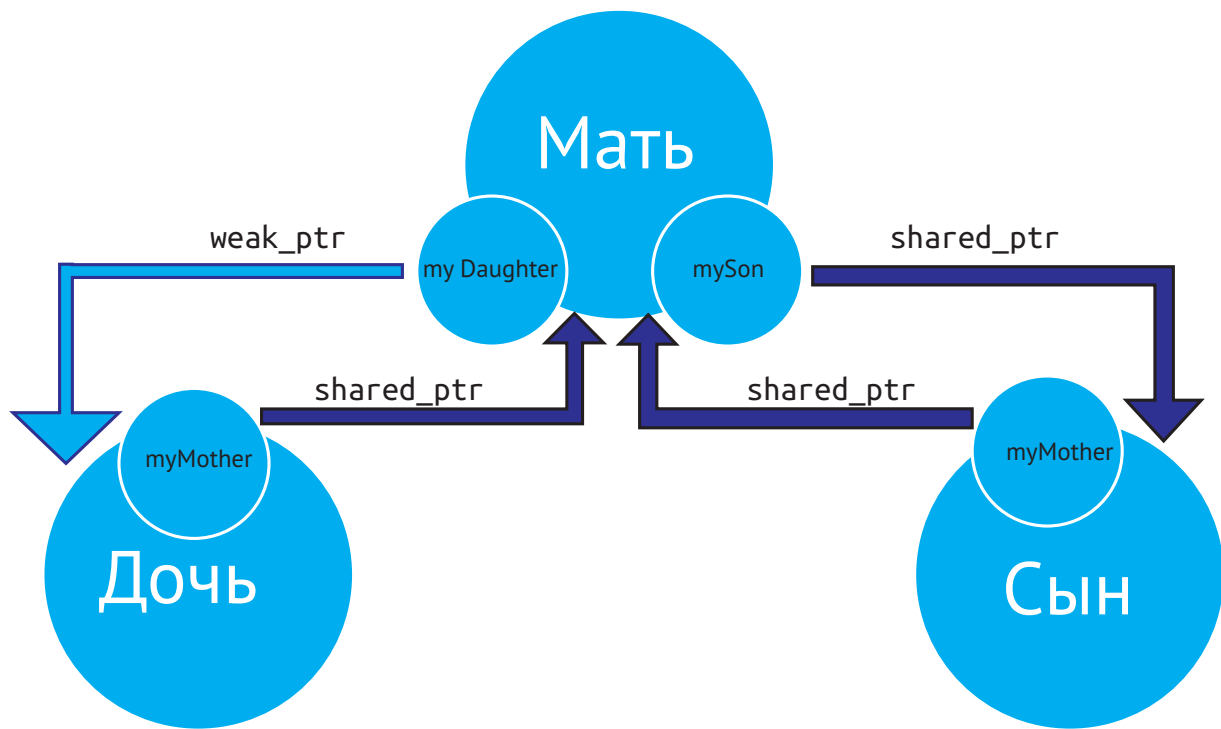
Имя	Описание
<code>expired</code>	Проверяет, не был ли ресурс удален
<code>lock</code>	Создает коллективный умный указатель <code>std::shared_ptr</code> на ресурс, на который указывает слабый указатель
<code>reset</code>	Сбрасывает ресурс
<code>swap</code>	Меняет ресурсы местами
<code>use_count</code>	Возвращает значение счетчика ссылок

Слабый указатель `std::weak_ptr` существует по одной причине: он разрывает цикл коллективного умного указателя `std::shared_ptr`.

Циклические ссылки

Циклические ссылки коллективного умного указателя `std::shared_ptr` возникают, когда они ссылаются друг на друга, вследствие чего счетчик ресурсов никогда не становится равным 0, и ресурс не будет автоматически высвобожден. Этот цикл можно разрывать, если встраивать слабый указатель `std::weak_ptr` в цикл. Слабый указатель `std::weak_ptr` не изменяет счетчик ссылок.

В результате работы примера исходного кода автоматически высвобождается дочь, но не сын и не мать. Мать ссылается на сына через коллективный указатель `std::shared_ptr`, а на дочь – через слабый указатель `std::weak_ptr`. Рисунок помогает увидеть структуру исходного кода.



Циклические ссылки

Наконец, ниже приведен исходный код.

Циклические ссылки

```
// cyclicReference.cpp
...
#include <memory>
...
using namespace std;
```

```
struct Son, Daughter;

struct Mother{
    ~Mother(){cout << "Mother gone" << endl;}
    void setSon(const shared_ptr<Son> s ){mySon= s;}
    void setDaughter(const shared_ptr<Daughter> d){myDaughter= d;}
    shared_ptr<const Son> mySon;
    weak_ptr<const Daughter> myDaughter;
};

struct Son{
    Son(shared_ptr<Mother> m):myMother(m){}
    ~Son(){cout << "Son gone" << endl;}
    shared_ptr<const Mother> myMother;
};

struct Daughter{
    Daughter(shared_ptr<Mother> m):myMother(m){}
    ~Daughter(){cout << "Daughter gone" << endl;}
    shared_ptr<const Mother> myMother;
};

{
    shared_ptr<Mother> mother= shared_ptr<Mother>(new Mother);
    shared_ptr<Son> son= shared_ptr<Son>(new Son(mother) );
    shared_ptr<Daughter> daugh= shared_ptr<Daughter>(new Daughter(mother));
    mother->setSon(son);
    mother->setDaughter(daugh);
}

// Daughter gone
```

Признаки типов

Библиотека `type_traits`¹ для работы с признаками (чертами) типов позволяет проверять, сравнивать и изменять типы во время компиляции, вследствие чего во время исполнения программы не возникает никаких накладных расходов. Библиотека `type_traits` используется по двум причинам: оптимизация и правильность. Оптимизация, потому что возможности библиотеки `type_traits` по интроспекции позволяют автоматически выбирать более быстрый исходный код. Правильность, потому что можно указывать требования к исходному коду, которые будут проверяться во время компиляции.

¹ См. http://en.cppreference.com/w/cpp/header/type_traits.

Проверка информации о типе

С помощью библиотеки `type_traits` можно проверять категории первичных (примитивных) и составных типов. Атрибут `value` содержит результат.



Мощная пара в виде библиотеки `type_traits` и вспомогательной шаблонной функции `static_assert`

Библиотека `type_traits` и вспомогательная шаблонная функция `static_assert` являются двумя мощными инструментами, если их применять в паре. С одной стороны, функции библиотеки `type_traits` предоставляют информацию о типе во время компиляции; с другой – функция `static_assert` выполняет проверку этой информации во время компиляции. Указанные проверки не вызывают никаких накладных расходов во время исполнения.

```
#include <type_traits>

...
template <typename T> fac(T a){
    static_assert(std::is_integral<T>::value, "T not integral");
    ...
}
fac(10);
fac(10.1); // with T= double; T not integral
```

Компилятор GCC выходит из функции `fac(10.1)` при ее вызове. Во время компиляции выдается сообщение, что `T` имеет тип `double` и, следовательно, не является интегральным типом.

Категории первичных типов

У первичных типов имеется 14 категорий. Они являются полными и не пересекаются, и поэтому каждый тип является членом только одной категории. При проверке категории своего типа запрос не зависит от квалификаторов `const` и `volatile`.

```
template <class T> struct is_void;
template <class T> struct is_integral;
template <class T> struct is_floating_point;
template <class T> struct is_array;
template <class T> struct is_pointer;
template <class T> struct is_null_pointer;
template <class T> struct is_member_object_pointer;
template <class T> struct is_member_function_pointer;
template <class T> struct is_enum;
template <class T> struct is_union;
template <class T> struct is_class;
template <class T> struct is_function;
```

```
template <class T> struct is_lvalue_reference;
template <class T> struct is_rvalue_reference;
```

В следующих ниже примерах исходного кода показаны все категории первичных типов.

Все категории первичных типов

```
// typeCategories.cpp
...
#include <type_traits>
using std::cout;

cout << std::is_void<void>::value;           // true
cout << std::is_integral<short>::value;      // true
cout << std::is_floating_point<double>::value; // true
cout << std::is_array<int [] >::value;       // true
cout << std::is_pointer<int*>::value;         // true
cout << std::is_reference<int&>::value;       // true

struct A{
    int a;
    int f(int){ return 2011; }
};

cout << std::is_member_object_pointer<int A::*>::value; // true
cout << std::is_member_function_pointer<int (A::*)(int)>::value; // true

enum E{
    e= 1,
};
cout << std::is_enum<E>::value;               // true

union U{
    int u;
};
cout << std::is_union<U>::value;              // true

cout << std::is_class<std::string>::value;     // true
cout << std::is_function<int * (double)>::value; // true
cout << std::is_lvalue_reference<int&>::value;  // true
cout << std::is_rvalue_reference<int&&>::value; // true
```

Категории составных типов

У составных типов имеется 7 категорий, которые основаны на 14 категориях первичных типов.

Таблица. Категории составных типов

Категория составного типа	Категория первичного типа
<code>is_arithmetic</code>	<code>is_floating_point</code> или <code>is_integral</code>
<code>is_fundamental</code>	<code>is_arithmetic</code> или <code>is_void</code>
<code>is_object</code>	<code>is_arithmetic</code> или <code>is_enum</code> или <code>is_pointer</code> или <code>is_member_pointer</code>
<code>is_reference</code>	<code>is_lvalue_reference</code> или <code>is_rvalue_reference</code>
<code>is_compound</code>	дополнение типа <code>is_fundamental</code>
<code>is_member_pointer</code>	<code>is_member_object_pointer</code> или <code>is_member_function_pointer</code>
<code>is_scalar</code>	<code>is_arithmetic</code> или <code>is_enum</code> или <code>is_pointer</code> или <code>is_is_member_pointer</code> или <code>is_null_pointer</code>

Свойства типов

Помимо категорий первичных и составных типов, имеется ряд свойств типов.

```

template <class T> struct is_const;
template <class T> struct is_volatile;
template <class T> struct is_trivial;
template <class T> struct is_trivially_copyable;
template <class T> struct is_standard_layout;
template <class T> struct has_unique_object_representation;
template <class T> struct is_empty;
template <class T> struct is_polymorphic;
template <class T> struct is_abstract;
template <class T> struct is_final;
template <class T> struct is_aggregate;
template <class T> struct is_implicit_lifetime;

template <class T> struct is_signed;
template <class T> struct is_unsigned;

template <class T> struct is_bounded_array;
template <class T> struct is_unbounded_array;
template <class T> struct is_scoped_enum;

template <class T, class... Args> struct is_constructible;
template <class T> struct is_default_constructible;
template <class T> struct is_copy_constructible;
template <class T> struct is_move_constructible;

template <class T, class U> struct is_assignable;

```

```

template <class T> struct is_copy_assignable;
template <class T> struct is_move_assignable;
template <class T> struct is_destructible;
template <class T, class... Args> struct is_trivially_constructible;
template <class T> struct is_trivially_default_constructible;
template <class T> struct is_trivially_copy_constructible;
template <class T> struct is_trivially_move_constructible;
template <class T, class U> struct is_trivially_assignable;
template <class T> struct is_trivially_copy_assignable;
template <class T> struct is_trivially_move_assignable;

template <class T> struct is_trivially_destructible;

template <class T, class... Args> struct is_nothrow_constructible;
template <class T> struct is_nothrow_default_constructible;
template <class T> struct is_nothrow_copy_constructible;
template <class T> struct is_nothrow_move_constructible;

template <class T, class U> struct is_nothrow_assignable;
template <class T> struct is_nothrow_copy_assignable;
template <class T> struct is_nothrow_move_assignable;

template <class T> struct is_nothrow_destructible;
template <class T> struct has_virtual_destructor;

template <class T> struct is_swappable_with;
template <class T> struct is_swappable;
template <class T> struct is_nothrow_swappable_with;
template <class T> struct is_nothrow_swappable;

```

Отношения типов

Библиотека `type_traits` поддерживает различные виды отношений типов.

Таблица. Отношения типов

Отношение	Описание
template <class Base, class Derived> struct is_base_of	Проверяет, не является ли тип Derived производным от типа Base
template <class From, class To> struct is_convertible struct is_nothrow_convertible	Проверяет, может ли тип From быть преобразован в тип To
template <class T, class U> struct is_same	Проверяет одинаковость типов T и U

Отношение	Описание
template <class T, class U> struct is_layout_compatible	Проверяет совместимость типов T и U по компоновке ¹
template <class Base, class Derived> struct is_pointer_interconvertible_base_of	Проверяет, не является ли тип указателем – межконвертируемой базой другого типа
template <class Fn, class ... ArgTypes> struct is_invocable struct is_invocable_r struct is_nothrow_invocable struct is_nothrow_invocable_r	Проверяет, может ли тип вызываться с заданными типами

Модификации типов

Библиотека `type_traits` позволяет модифицировать типы во время компиляции. При этом можно изменять константность (`const`) типа:

Модификации типов

```
// typeTraitsModifications.cpp
...
#include <type_traits>
...
using namespace std;

cout << is_const<int>::value;           // false
cout << is_const<const int>::value;      // true
cout << is_const<add_const<int>::type>::value; // true

typedef add_const<int>::type myConstInt;
cout << is_const<myConstInt>::value;      // true

typedef const int myConstInt2;
cout << is_same<myConstInt, myConstInt2>::value; // true

cout << is_same<int, remove_const<add_const<int>::type>::type>::value; // true
cout << is_same<const int, add_const<add_const<int>::type>::type>::value; // true
```

Шаблонная структура `std::add_const` добавляет константность к типу, а шаблонная структура `std::remove_const` удаляет ее.

В библиотеке `type_traits` имеется ряд других функциональностей по работе с признаками типов. Например, с помощью следующих ниже шаблонных структур признаков типов можно модифицировать константно-волатильные свойства типа.

¹ См. https://en.cppreference.com/w/cpp/language/data_members#Standard_layout.

```
template <class T> struct remove_const;
template <class T> struct remove_volatile;
template <class T> struct remove_cv;
```

```
template <class T> struct add_const;
template <class T> struct add_volatile;
template <class T> struct add_cv;
```

Можно изменять знак во время компиляции:

```
template <class T> struct make_signed;
template <class T> struct make_unsigned;
```

либо свойства типа, связанные с ссылкой или указателем:

```
template <class T> struct remove_reference;
template <class T> struct remove_cvref;
template <class T> struct add_lvalue_reference;
template <class T> struct add_rvalue_reference;
```

```
template <class T> struct remove_pointer;
template <class T> struct add_pointer;
```

Следующие ниже шаблонные структуры признаков типов бесценны для написания обобщенных библиотек.

```
template <class B> struct enable_if;
template <class B, class T, class F> struct conditional;
template <class... T> common_type;
template <class... T> common_reference;
template <class... T> basic_common_reference;
template <class... T> void_t;
template <class... T> type_identity;
```

С помощью шаблонной структуры `std::enable_if` можно условно скрывать перегрузку функции или конкретизацию шаблона от выявления наиболее подходящей перегрузки. Шаблонная структура `std::conditional` предоставляет трехместный оператор во время компиляции, а шаблонная структура `std::common_type` дает общий тип всех типов. Шаблонные структуры `std::common_type`, `std::common_reference`, `std::basic_common_reference`, `std::void_t` и `std::type_identity` являются вариативными шаблонами¹, вследствие чего количество параметров типов может быть произвольным.

Операции на признаках

Шаблонные структуры `std::conjunction`, `std::disjunction` и `std::negation` позволяют логически комбинировать функциональности признаков типов. Они являются вариативными шаблонами.

¹ См. http://en.cppreference.com/w/cpp/language/parameter_pack.



В языке C++ есть сокращенное написание для `::type` и для `::value`

Если требуется получить `const int` из `int`, то необходимо запросить тип: `std::add_const<int>::type`. В стандарте C++14 вместо многословной формы `std::add_const<int>::type` рекомендуется использовать просто `std::add_const_t<int>`. Это правило работает для всех функциональностей признаков типов.

Соответственно, в стандарте C++17 можно использовать сокращенное написание `std::is_integral_v<T>` для предиката `std::is_integral<T>::value`.

Отношения между членами

Шаблонная структура `std::is_pointer_interconvertible_with_class` проверяет, можно ли взаимоконвертировать указатели на объекты некоего типа и указатель на конкретизированный подобъект этого типа, а шаблонная структура `std::is_corresponding_member` проверяет, соответствуют ли друг другу два конкретизированных члена в общей начальной подпоследовательности двух конкретизированных типов.

Константно-вычисляемый контекст

Вызов стандартной функции `std::is_constant_evaluated` позволяет обнаруживать, не происходит ли вызов определенной функции во время компиляции.

Обнаружение вызова функции во время компиляции

```
// constantEvaluated.cpp
#include <type_traits>
...

constexpr double power(double b, int x) {
    if (std::is_constant_evaluated() && !(b == 0.0 && x < 0)) {

        if (x == 0)
            return 1.0;
        double r = 1.0, p = x > 0 ? b : 1.0 / b;
        auto u = unsigned(x > 0 ? x : -x);
        while (u != 0) {
            if (u & 1) r *= p;
            u /= 2;
            p *= p;
        }
        return r;
    }
}
```

```
    else {  
        return std::pow(b, double(x));  
    }  
}  
  
constexpr double kilo1 = power(10.0, 3); // исполнение во время компиляции  
  
int n = 3;  
double kilo2 = power(10.0, n); // исполнение во время работы программы  
std::cout << "kilo2: " << kilo2 << '\n';
```

Библиотека для работы со временем

Библиотека `chrono` для работы со временем¹ состоит из трех главных компонентов: временной точки, продолжительности времени и часов. В дополнение к этому указанная библиотека обеспечивает функциональность времени суток, поддержку календаря, часовых поясов и ввода-вывода.

Временная точка

Временная точка задается начальной точкой, так называемой эпохой, и дополнительной продолжительностью времени.

Продолжительность времени

Продолжительность времени – это разница между двумя временными точками. Она задается количеством тиков.

Часы

Часы состоят из начальной точки (эпохи) и тика, по которому можно вычислить текущую временную точку.

Время суток

Время суток – это время, прошедшее с полуночи, деленное на часы:минуты:секунды.

Календарь

Календарь обозначает различные календарные дни, такие как год, месяц, будний день или n -й день недели.

Часовой пояс

Часовой пояс представляет время, характерное для определенной географической территории.

¹ См. <http://en.cppreference.com/w/cpp/header/chrono>.



Библиотека `chrono` – ключевой компонент работы в многопоточной среде

Библиотека `chrono` является ключевым компонентом новых способностей языка C++ по многопоточной обработке. Например, текущий поток инструкций можно перевести в сон с помощью вызова `std::this_thread::sleep_for(std::chrono::milliseconds(15))` на 15 миллисекунд или попытаться приобрести блокировочный замок на 2 минуты: `lock.try_lock_until(now + std::chrono::minutes(2))`.

Временная точка

Продолжительность состоит из периода времени, определяемого как некоторое количество тиков некоторой единицы времени. Временная точка состоит из часов и продолжительности времени. Эта продолжительность может быть положительной либо отрицательной.

```
template <class Clock, class Duration= typename Clock::duration>
class time_point;
```

Для часов `std::chrono::steady_clock`, `std::chrono::high_resolution_clock` и `std::chrono::system` эпоха не определена. Но на популярной платформе эпоха `std::chrono::system` обычно определяется как 1.1.1970. Время рассчитывается начиная с 1.1.1970 с глубинами разрешения, равными наносекундам, секундам и минутам.

Время, прошедшее с эпохи

```
// epoch.cpp
...
#include <chrono>
...
auto timeNow= std::chrono::system_clock::now();
auto duration= timeNow.time_since_epoch();
std::cout << duration.count() << "ns"           // 1413019260846652ns

typedef std::chrono::duration<double> MySecondTick;
MySecondTick mySecond(duration);
std::cout << mySecond.count() << "s";           // 1413019260.846652s

const int minute= 60;
typedef std::chrono::duration<double, <minute>> MyMinuteTick;
MyMinuteTick myMinute(duration);
std::cout << myMinute.count() << "m";           // 23550324.920572m
```

Благодаря вспомогательной шаблонной функции `std::chrono::clock_cast` можно выполнять приведение временных точек к различным часам.



Простые тесты на производительность с помощью библиотеки `chrono`

Измерение производительности

```
// performanceMeasurement.cpp
...
#include <chrono>
...
std::vector<int> myBigVec(10000000, 2011);
std::vector<int> myEmptyVec1;

auto begin= std::chrono::high_resolution_clock::now();
myEmptyVec1 = myBigVec;
auto end= std::chrono::high_resolution_clock::now() - begin;

auto timeInSeconds = std::chrono::duration<double>(end).count();
std::cout << timeInSeconds << '\n';    // 0.0150688800
```

Продолжительность времени

Продолжительность времени – это разница между двумя временными точками. Продолжительность времени измеряется в количестве тиков.

```
template <class Rep, class Period = ratio<1>> class duration;
```

Если `Rep` – это число с плавающей точкой, то продолжительность времени поддерживает доли тиков. В библиотеке `chrono` предопределены наиболее важные продолжительности времени:

```
typedef duration<signed int, nano>      nanoseconds;
typedef duration<signed int, micro>     microseconds;
typedef duration<signed int, milli>     milliseconds;
typedef duration<signed int>            seconds;
typedef duration<signed int, ratio< 60>> minutes;
typedef duration<signed int, ratio<3600>> hours;
```

Какова может быть продолжительность времени? Стандарт C++ гарантирует, что предопределенные продолжительности времени могут хранить ± 292 года. При этом имеется возможность легко определять свою собственную продолжительность времени, к примеру немецкий школьный час: `typedef std::chrono::duration<double, std::ratio<2700>> MyLessonTick`. Продол-

жительность времени в натуральных числах должна явно конвертироваться в продолжительность времени в числах с плавающей точкой. Значение будет усечено:

Продолжительности

```
// duration.cpp
...
#include <chrono>
#include <ratio>

using std::chrono;

typedef duration<long long, std::ratio<1>> MySecondTick;
MySecondTick aSecond(1);

milliseconds milli(aSecond);
std::cout << milli.count() << " milli";           // 1000 milli

seconds seconds(aSecond);
std::cout << seconds.count() << " sec";           // 1 sec

minutes minutes(duration_cast<minutes>(aSecond));
std::cout << minutes.count() << " min";           // 0 min

typedef duration<double, std::ratio<2700>> MyLessonTick;
MyLessonTick myLesson(aSecond);
std::cout << myLesson.count() << " less";         // 0.00037037 less
```

В стандарт C++14 встроены литералы для наиболее часто используемых временных интервалов.

Таблица. Встроенные литералы для продолжительности времени

Тип	Суффикс	Пример
std::chrono::hours	h	5h
std::chrono::minutes	min	5min
std::chrono::seconds	s	5s
std::chrono::milliseconds	ms	5ms
std::chrono::microseconds	us	5us
std::chrono::nanoseconds	ns	5ns



Вспомогательный шаблонный класс `std::ratio`

Вспомогательный шаблонный класс `std::ratio` поддерживает арифметические операции с рациональными числами во время компиляции. Рациональное число имеет два аргумента шаблона: числитель и знаменатель. В стандарте C++11 предопределен целый ряд рациональных чисел.

```
typedef ratio<1, 1000000000000000000> atto;
typedef ratio<1, 1000000000000000> femto;
typedef ratio<1, 1000000000000> pico;
typedef ratio<1, 1000000000> nano;
typedef ratio<1, 1000000> micro;
typedef ratio<1, 1000> milli;
typedef ratio<1, 100> centi;
typedef ratio<1, 10> deci;
typedef ratio< 10, 1> deca;
typedef ratio< 100, 1> hecto;
typedef ratio< 1000, 1> kilo;
typedef ratio< 1000000, 1> mega;
typedef ratio< 1000000000, 1> giga;
typedef ratio< 1000000000000, 1> tera;
typedef ratio< 1000000000000000, 1> peta;
typedef ratio< 1000000000000000000, 1> exa;
```

Компилятор GCC выходит из функции `fac(10.1)` при ее вызове. Во время компиляции выдается сообщение, что `T` имеет тип `double` и, следовательно, не является интегральным типом.

Часы

Часы состоят из точки отсчета и тика. Текущее время можно узнать с помощью функции-члена `now`.

`std::chrono::system_clock`

Класс, используемый для работы системными часами, которые можно синхронизировать с внешними часами.

`std::chrono::steady_clock`

Класс, представляющий часы, которые нельзя отрегулировать.

`std::chrono::high_resolution_clock`

Класс, используемый для работы с системными часами, имеющими наибольшую глубину точности.

Класс `std::chrono::system_clock` обычно ссылается на 1.1.1970. Класс `std::steady_clock` невозможно корректировать вперед либо назад, противоположно двум другим часам. Функции-члены `to_time_t` и `from_time_t` могут использоваться для конвертации между объектами классов `std::chrono::system_clock` и `std::time_t`.

Время дня

Класс `std::chrono::time_of_day` разбивает продолжительность времени, прошедшего с полуночи, на часы:минуты:секунды. Функции `std::chrono::is_am` и `std::chrono::is_pm` проверяют, не находится ли измеряемое время до полудня (*ante meridiem*) либо после полудня (*post meridiem*).

Объект `std::chrono::time_of_day` `tOfDay` поддерживает различные функции-члены.

Таблица. Функции-члены объекта `std::chrono::time_of_day` `tOfDay`

Функция-член	Описание
<code>tOfDay.hours()</code>	Возвращает часовую составляющую с полуночи
<code>tOfDay.minutes()</code>	Возвращает минутную составляющую с полуночи
<code>tOfDay.seconds()</code>	Возвращает секундную составляющую с полуночи
<code>tOfDay.subseconds()</code>	Возвращает дробную секундную составляющую с полуночи
<code>tOfDay.to_duration()</code>	Возвращает продолжительность времени с полуночи
<code>std::chrono::make12(hr)</code> <code>std::chrono::make24(hr)</code>	Возвращает 12-часовой (24-часовой) эквивалент времени в 24-часовом (12-часовом) формате
<code>std::chrono::is_am(hr)</code> <code>std::chrono::is_pm(hr)</code>	Распознает, не находится ли время, измеряемое в 24-часовом формате, до полудня либо после полудня

Календарь

Календарь обозначает различные календарные даты, такие как год, месяц, день недели или *n*-й день недели.

Текущее время

```
// currentTime.cpp
...
#include <chrono>
using std::chrono;
...
auto now = system_clock::now();
std::cout << "The current time is " << now << " UTC\n";

auto currentYear = year_month_day(floor<days>(now)).year();
std::cout << "The current year is " << currentYear << "\n";

auto h = floor<hours>(now) - sys_days(January/1/currentYear);
std::cout << "It has been " << h << " since New Years!\n";

std::cout << "\n";
```

```
auto birthOfChrist = year_month_weekday(sys_days(January/01/0000));
std::cout << "Weekday: " << birthOfChrist.weekday() << '\n';
```

Результат работы приведенной выше программы показывает информацию, относящуюся к текущему времени.

```
The current time is 2020-07-18 20:39:12.356023527 UTC
The current year is 2020
It has been 4796h since New Years!

Weekday: Sat
```

В следующей ниже таблице приведен краткий обзор календарных типов.

Таблица. Различные календарные типы

Класс	Описание
<code>last_spec</code>	Указывает последний день либо будний день в месяце
<code>day</code>	Представляет день месяца
<code>month</code>	Представляет месяц года
<code>year</code>	Представляет год по григорианскому календарю
<code>weekday</code>	Представляет день недели по григорианскому календарю
<code>weekday_indexed</code>	Представляет <i>n</i> -й будний день месяца
<code>weekday_last</code>	Представляет последний будний день месяца
<code>month_day</code>	Представляет конкретный день конкретного месяца
<code>month_day_last</code>	Представляет последний день конкретного месяца
<code>month_weekday</code>	Представляет <i>n</i> -й будний день конкретного месяца
<code>month_weekday_last</code>	Представляет последний будний день конкретного месяца
<code>year_month</code>	Представляет конкретный месяц конкретного года
<code>year_month_day</code>	Представляет конкретный год, месяц и день
<code>year_month_day_last</code>	Представляет последний день конкретного года и месяца
<code>year_month_weekday</code>	Представляет последний день конкретного года и месяца
<code>year_month_weekday_last</code>	Представляет последний будний день конкретного года и месяца

Часовой пояс

Часовые пояса представляют собой время, характерное для конкретной географической территории. Следующий ниже фрагмент программы показывает местное время в различных часовых поясах.

Вывод местного времени в различных часовых поясах

```
// timezone.cpp
...
#include <chrono>
using std::chrono;
..
auto time = floor<milliseconds>(system_clock::now());
auto localTime = zoned_time<milliseconds>(current_zone(), time);
auto berlinTime = zoned_time<milliseconds>("Europe/Berlin", time);
auto newYorkTime = std::chrono::zoned_time<milliseconds>("America/New_York",
time);
auto tokyoTime = zoned_time<milliseconds>("Asia/Tokyo", time);

std::cout << time << '\n';
std::cout << localTime << '\n';
std::cout << berlinTime << '\n';
std::cout << newYorkTime << '\n';
std::cout << tokyoTime << '\n';
```

Функциональность часового пояса поддерживает доступ к базе данных часовых поясов IANA¹, позволяет работать с различными часовыми поясами и предоставляет информацию о високосных секундах.

В следующей ниже таблице приведен краткий обзор функциональности часового пояса. Более подробную информацию можно найти на веб-сайте en.cppreference.com².

Таблица. Информация о часовом поясе

Тип	Описание
tzdb	Описывает базу данных часовых поясов IANA
locate_zone	Находит часовой пояс по его имени
current_zone	Возвращает текущий часовой пояс
time_zone	Представляет часовой пояс
sys_info	Возвращает информацию о часовом поясе в конкретный момент времени
local_info	Представляет информацию о местном времени в соответствии с процедурами конвертации времени UNIX
zoned_time	Представляет часовой пояс и временную точку
leap_second	Содержит информацию о вставке високосной секунды

¹ См. <https://www.iana.org/time-zones>.

² См. <https://en.cppreference.com/w/cpp/chrono>.

Функциональность библиотеки `chrono` по вводу-выводу

Вспомогательная функция `std::chrono::parse` вычленяет объект `chrono` из потока и выполняет его структурный разбор.

Разбор временной точки и часового пояса

```
std::istringstream inputStream{"1999-10-31 01:30:00 -08:00 US/Pacific"};

std::chrono::local_seconds timePoint;
std::string timeZone;
inputStream >> std::chrono::parse("%F %T %Ez %Z", timePoint, timeZone);
```

Функциональность структурного разбора предоставляет различные конкретизаторы формата для работы с временем суток и календарными датами, такими как год, месяц, неделя и день. Веб-сайт cppreference.com¹ предоставляет подробную информацию о конкретизаторах формата.

Новые обобщенные типы `std::any`, `std::optional` и `std::variant`

В стандарте C++17 появились новые обобщенные типы данных `std::any`, `std::optional` и `std::variant`, которые основаны на библиотеках Boost².

`std::any`

Обобщенный тип `std::any`³ – это типобезопасный контейнер для одиночных значений любого типа, конструируемого путем копирования. Для этого типа требуется заголовок `<any>`. Контейнер `std::any` можно создавать несколькими способами. Например, можно использовать различные конструкторы либо фабричную функцию `std::make_any`. Функция-член `any.emplace` позволяет напрямую конструировать одно значение в контейнере `any`. Функция-член `any.reset` позволяет уничтожать содержащийся объект. Если требуется узнать о наличии значения в контейнере `any`, то рекомендуется использовать функцию-член `any.has_value`. Более того, посредством функции-члена `any.type` можно даже получать оператор `typeid` контейнерного объекта. Благодаря обобщенной функции `std::any_cast` можно обращаться к содержащемуся объекту. Если указывается неправильный тип, то генерируется исключение `std::bad_any_cast`. Ниже приведен фрагмент исходного кода, демонстрирующий базовое использование объекта обобщенного типа `std::any`.

¹ См. <https://en.cppreference.com/w/cpp/chrono/parse>.

² См. <http://www.boost.org/>.

³ См. <http://en.cppreference.com/w/cpp/utility/any>.

std::any

```
// any.cpp
...
#include <any>

struct MyClass{};

...

std::vector<std::any> anyVec{true, 2017, std::string("test"), 3.14, MyClass()};
std::cout << std::any_cast<bool>(anyVec[0]);    // true
int myInt= std::any_cast<int>(anyVec[1]);
std::cout << myInt << '\n';                    // 2017

std::cout << anyVec[0].type().name();           // b
std::cout << anyVec[1].type().name();           // i
```

В приведенном выше фрагменте программы определен контейнер `std::vector<std::any>`. Для того чтобы получить один из его элементов, необходимо использовать обобщенную функцию `std::any_cast`. Как уже говорилось ранее, при применении неправильного типа генерируется исключение `std::bad_any_cast`.

**Строковое представление оператора typeid**

Строковое представление оператора `typeid` определяется реализацией. Если `anyVec[1]` имеет тип `int`, то в компиляторе GCC C++¹ выражение `anyVec[1].type().name()` вернет `i`, а в компиляторе Microsoft Visual C++² оно вернет `int`.

Объект типа `std::any` может иметь объекты произвольных типов; объект типа `std::optional` может иметь значение либо не иметь его.

std::optional

Обобщенный тип `std::optional`³ удобен для вычислений, таких как запросы к базе данных, которые, возможно, будут иметь результат. Для этого типа требуется заголовок `<optional>`.

Различные конструкторы и вспомогательная шаблонная функция `std::make_optional` позволяют задавать опциональный объект `opt` со значением либо без него. Функция-член `opt.emplace` конструирует содержащееся в нем значение прямо на месте, а функция-член `opt.reset` уничтожает значение контейнера. К контейнеру `std::optional` можно обращаться в явной форме с запросом о на-

¹ См. <https://gcc.gnu.org/>.

² См. https://en.wikipedia.org/wiki/Microsoft_Visual_C%2B%2B.

³ См. <http://en.cppreference.com/w/cpp/utility/optional>.

личии у него значения или проверять его в логическом выражении. Функция-член `opt.value` возвращает значение, а функция-член `opt.value_or` возвращает значение либо дефолтное значение в противном случае. Если объект `opt` не содержит значения, то вызов функции-члена `opt.value` генерирует исключение `std::bad_optional_access`.



Для обозначения отсутствующего результата не рекомендуется использовать обычные значения

До появления стандарта C++17 было принято использовать уникальное значение, такое как указатель `null`, пустой строковый литерал либо уникальное целое число, чтобы обозначать отсутствующий результат. Говоря о системе типов, для определения необычного значения приходится задействовать обычное значение, например пустой строковый литерал. Такие уникальные значения или отсутствующий результат чреваты ошибками, так как систему типов приходится использовать ненадлежащим образом для проверки возвращаемого значения.

Ниже приведен небольшой пример с использованием объекта обобщенного типа `std::optional`.

```
std::optional
```

```
// optional.cpp
...
#include <optional>

std::optional<int> getFirst(const std::vector<int>& vec){
    if (!vec.empty()) return std::optional<int>(vec[0]);
    else return std::optional<int>();
}

...

std::vector<int> myVec{1, 2, 3};
std::vector<int> myEmptyVec;

auto myInt= getFirst(myVec);

if (myInt){
    std::cout << *myInt << '\n';           // 1
    std::cout << myInt.value() << '\n';     // 1
    std::cout << myInt.value_or(2017) << '\n'; // 1
}

auto myEmptyInt= getFirst(myEmptyVec);
```

```
if (!myEmptyInt){
    std::cout << myEmptyInt.value_or(2017) << '\n'; // 2017
}
```

Объект типа `std::optional` используется в функции `getFirst`. Функция `getFirst` возвращает первый элемент, если он существует. В противном случае будет получен объект `std::optional<int>`. В главной функции находятся два вектора. На обоих векторах вызывается функция `getFirst`, которая возвращает объект типа `std::optional`. В случае переменной `myInt` объект имеет значение, а в случае переменной `myEmptyInt` – нет. Программа выводит на консоль значения `myInt` и `myEmptyInt`. Вызов функции-члена `myInt.value_or(2017)` возвращает значение, а вызов функции-члена `myEmptyInt.value_or(2017)` возвращает дефолтное значение.

В стандарте C++23 обобщенный тип `std::optional` расширен монадическими операциями `opt.and_then`, `opt.transform` и `opt.or_else`. Функция-член `opt.and_then` возвращает результат вызова заданной функции, если он существует, либо пустой объект `std::optional`. Функция-член `opt.transform` возвращает объект типа `std::optional`, содержащий преобразованное значение либо пустой объект типа `std::optional`. Вдобавок функция-член `opt.or_else` возвращает объект типа `std::optional`, если он содержит значение, либо в противном случае возвращает результат заданной функции.

Указанные монадические операции позволяют собирать операции в композиции на объектах типа `std::optional`:

Монадические операции на объекте обобщенного типа `std::optional`

```
// optionalMonadic.cpp

#include <iostream>
#include <optional>
#include <vector>
#include <string>

std::optional<int> getInt(std::string arg) {
    try {
        return {std::stoi(arg)};
    }
    catch (...) {
        return { };
    }
}

int main() {

    std::vector<std::optional<std::string>> strings = {"66", "foo", "-5"};
```

```
for (auto s: strings) {  
    auto res = s.and_then(getInt)  
        .transform( [](int n) { return n + 100; })  
        .transform( [](int n) { return std::to_string(n); })  
        .orElse([] { return std::optional{std::string("Error") }; });  
    std::cout << *res << ' ';           // 166 Error 95  
}  
  
}
```

Диапазонный цикл `for` (строка 22) выполняет итерации по контейнеру `std::vector<std::optional<std::string>>`. Сначала функция `getInt` конвертирует каждый элемент в целое число (строка 23), прибавляет к нему 100 (строка 24), снова конвертирует в строковый объект (строка 25) и, наконец, выводит содержимое строкового объекта на консоль (строка 27). Если первоначальная конвертация в `int` не удастся, то возвращается строковый литерал "Error" (строка 26), который выводится на консоль.

std::variant

Обобщенный тип `std::variant`¹ – это типобезопасное объединение. Для этого типа требуется заголовок `<variant>`. Экземпляр обобщенного типа `std::variant` имеет значение одного из своих типов. Тип не должен быть ссылкой, массивом либо `void`. Объект типа `std::variant` может иметь тип более одного раза. По умолчанию объект типа `std::variant` инициализируется своим первым типом, поэтому его первый тип должен иметь дефолтный конструктор. С помощью функции-члена `var.index` можно получать индекс альтернативы с отсчетом от нуля, которую содержит объект `var` обобщенного типа `std::variant`. Функция-член `var.valueless_by_exception` возвращает `false`, если вариант содержит значение. С помощью функции-члена `var.emplace` можно создавать новое значение прямо на месте. Для доступа к объекту типа `std::variant` используется несколько функций-членов. Функция-член `var.holds_alternative` позволяет проверять наличие у объекта типа `std::variant` указанной альтернативы. Вспомогательную функцию `std::get` можно использовать с индексом и типом в качестве аргумента. При использовании индекса будет получено значение. Если вспомогательная шаблонная функция `std::get` вызывается с типом, то значение будет получено, только если оно является уникальным. При использовании недопустимого индекса или неуникального типа генерируется исключение `std::bad_variant_access`. В отличие от вспомогательной шаблонной функции `std::get`, которая в случае ошибки возвращает исключение, вспомогательная шаблонная функция `std::get_if` в таком случае возвращает указатель `null`.

Следующий ниже фрагмент исходного кода демонстрирует использование объекта обобщенного типа `std::variant`.

¹ <http://en.cppreference.com/w/cpp/utility/variant>.

std::variant

```

// variant.cpp
...
#include <variant>

...

std::variant<int, float> v, w;
v = 12; // v содержит int
int i = std::get<int>(v);
w = std::get<int>(v);
w = std::get<0>(v); // тот же эффект, что и в предыдущей строке
w = v; // тот же эффект, что и в предыдущей строке

// std::get<double>(v); // ошибка: нет double в [int, float]
// std::get<3>(v); // ошибка: допустимые индексные значения – это 0 и 1

try{
std::get<float>(w); // w содержит int, не float: выдаст исключение
}
catch (std::bad_variant_access&) {}
std::variant<std::string> v2("abc"); // конвертирующий конструктор
// должен быть однозначным
v2 = "def"; // конвертирующее присваивание
// должно быть однозначным

```

`v` и `w` – это два варианта. Оба могут иметь значение типа `int` и типа `float`. По умолчанию их значение равно 0. Вариант `v` становится равным 12, и следующий вызов вспомогательной шаблонной функции `std::get<int>(v)` возвращает это значение. В следующих трех строках показаны три возможности присваивать вариант `v` варианту `w`, но при этом необходимо помнить несколько правил. Значение варианта можно запрашивать по типу: `std::get<double>(v)` – либо по индексу: `std::get<3>(v)`. Тип должен быть уникальным, а индекс – допустимым. Вариант `w` содержит значение типа `int`; поэтому при запросе типа `float` будет сгенерировано исключение `std::bad_variant_access`. Если вызов конструктора или вызов присваивания однозначен, то конвертация может произойти. Это позволяет построить объект `std::variant<std::string>` из последовательности символов `S` либо присвоить варианту новую последовательность символов `S`.

В обобщенном типе `std::variant` есть интересная функция-член `std::visit`, которая позволяет исполнять вызываемую единицу кода на списке вариантов. Вызываемая единица кода – это блок кода, который можно вызывать. Обычно это может быть функция, функциональный объект либо лямбда-выражение. Ради простоты в приведенном ниже примере используется лямбда-функция.

std::visit

```

// visit.cpp
...
#include <variant>

...

std::vector<std::variant<char, long, float, int, double, long long>>
    vecVariant = {5, '2', 5.4, 100ll, 2011l, 3.5f, 2017};

for (auto& v: vecVariant){
    std::visit([](auto&& arg){std::cout << arg << " ";}, v);
    // 5 2 5.4 100 2011 3.5 2017
}

// показать все типы
for (auto& v: vecVariant){
    std::visit([](auto&& arg){std::cout << typeid(arg).name() << " ";}, v);
    // int char double __int64 long float int
}

// получить сумму
std::common_type<char, long, float, int, double, long long>::type res{};

std::cout << typeid(res).name() << '\n'; // double

for (auto& v: vecVariant){
    std::visit([&res](auto&& arg){res+= arg;}, v);
}
std::cout << "res: " << res << '\n';    // 4191.9

// удвоить каждое значение
for (auto& v: vecVariant){
    std::visit([&res](auto&& arg){arg *= 2;}, v);
    std::visit([](auto&& arg){std::cout << arg << " ";}, v);
    // 10 d 10.8 200 4022 7 4034
}

```

Каждый вариант в этом примере может содержать типы char, long, float, int, double или long long. Первый посетитель [](auto&& arg){std::cout << arg << " ";} выводит на консоль различные варианты. Второй посетитель std::cout << typeid(arg).name() << " ";} выводит его типы.

Далее элементы вариантов суммируются. Во-первых, нужен правильный тип результата во время компиляции. Его обеспечивает шаблонная структура

`std::common_type` из библиотеки `type_traits`. Она дает тип, в который можно неявно конвертировать все типы `char`, `long`, `float`, `int`, `double` и `long long`. Итоговые скобки `{}` в `res{}` инициализируют его значением `0.0`. Выражение `res` имеет тип `double`. Посетитель `[&res](auto&& arg){arg *= 2;}` вычисляет сумму, а следующая строка выводит ее на консоль.

`std::expected`

Вспомогательный шаблонный класс `std::expected<T, E>` предоставляет возможность хранить любое из двух значений. Экземпляр класса `std::expected` всегда содержит значение: ожидаемое значение типа `T` либо неожиданное значение типа `E`. Для этого типа требуется заголовок `<expected>`. Благодаря вспомогательному шаблонному классу `std::expected` можно реализовывать функции, которые возвращают значение либо ошибку. Сохраненное значение размещается непосредственно в памяти, занимаемой объектом `expected`. Никакого динамического выделения памяти не происходит.

Вспомогательный шаблонный класс `std::expected` имеет интерфейс, аналогичный обобщенному типу `std::optional`. В отличие от обобщенного типа `std::optional`, класс `std::expected` может возвращать сообщение об ошибке.

Различные конструкторы позволяют задавать объект `exp` с ожидаемым значением. Функция-член `exp.emplace` конструирует содержащееся в нем значение прямо на месте. К контейнеру `std::expected` можно делать запросы в явной форме о наличии у него значения либо проверять его в логическом выражении. Функция-член `exp.value` возвращает ожидаемое значение, а функция-член `exp.value_or` – ожидаемое значение либо дефолтное значение. Если объект `exp` имеет неожиданное значение, то вызов функции-члена `exp.value` генерирует исключение `std::bad_expected_access`.

Структура `std::unexpected`, используемая в контексте объекта класса `std::expected<T, E>`, представляет неожиданное значение, хранящееся в этом объекте.

`std::expected`

```
// expected.cpp

#include <iostream>
#include <expected>
#include <vector>
#include <string>

std::expected<int, std::string> getInt(std::string arg) {
    try {
        return std::stoi(arg);
    }
    catch (...) {
        return std::unexpected{std::string(arg + ": Error")};
    }
}
```

```
}

int main() {

    std::vector<std::string> strings = {"66", "foo", "-5"};

    for (auto s: strings) {
        auto res = getInt(s);
        if (res) {
            std::cout << res.value() << ' '; // 66 -5
        }
        else {
            std::cout << res.error() << ' '; // foo: Error
        }
    }

    std::cout << '\n';

    for (auto s: strings) {
        auto res = getInt(s);
        std::cout << res.value_or(2023) << ' '; // 66 2023 -5
    }

}
```

Функция `getInt` конвертирует каждый строковый объект в целое число и возвращает объект `std::expected<int, std::string>`. Тип `int` представляет ожидаемое, а строковый объект `std::string` – неожиданное значение. Два диапазонных цикла `for` (строки 22 и 34) выполняют итерации по контейнеру `std::vector<std::string>`. В первом цикле `for` (строка 22) на консоль выводится ожидаемое (строка 25) либо неожиданное значение (строка 28). Во втором цикле `for` (строка 34) выводится ожидаемое либо дефолтное значение 2023 (строка 36).

Для удобства композиции функций класс `std::expected` поддерживает монадические операции `exp.and_then`, `exp.transform`, `exp.or_else` и `exp.transform_error`. Функция-член `exp.and_then` возвращает результат вызова данной функции, если он существует, либо пустой объект класса `std::expected`. Функция-член `exp.transform` возвращает объект класса `std::expected`, содержащий преобразованное значение, либо пустой объект класса `std::expected`. Вдобавок функция-член `exp.or_else` возвращает объект класса `std::expected`, если он содержит значение, либо результат заданной функции в противном случае.

Следующая ниже программа основана на предыдущей программе `optionalMonadic.cpp`. Обобщенный тип `std::optional`, по сути дела, заменен на вспомогательный шаблонный класс `std::expected`.

Монадические операции на объекте класса `std::expected`

```
// expectedMonadic.cpp
...
#include <expected>

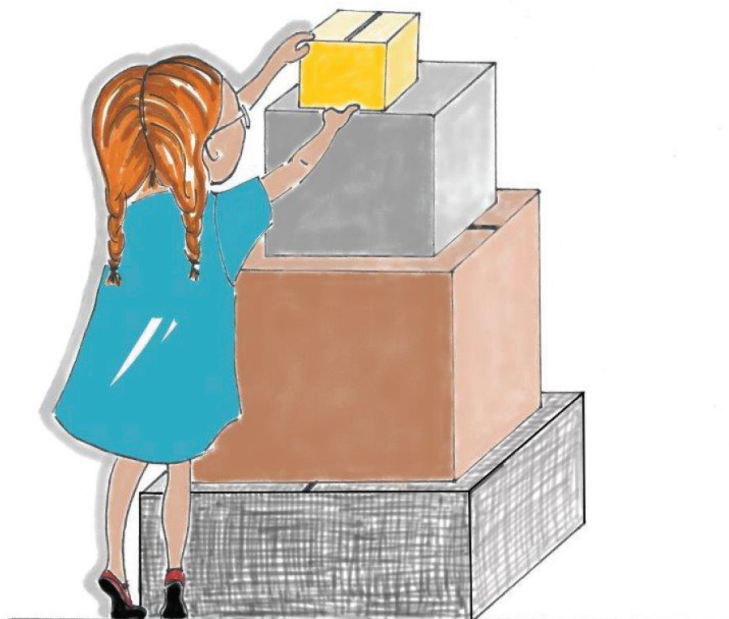
std::expected<int, std::string> getInt(std::string arg) {
    try {
        return std::stoi(arg);
    }
    catch (...) {
        return std::unexpected{std::string(arg + ": Error")};
    }
}

std::vector<std::string> strings = {"66", "foo", "-5"};

for (auto s: strings) {
    auto res = getInt(s)
        .transform( [](int n) { return n + 100; })
        .transform( [](int n) { return std::to_string(n); });
    std::cout << *res << ' '; // 166 foo: Error 95
}
```

Диапазонный цикл `for` (строка 23) выполняет итерации по контейнеру `std::vector<std::string>`. Сначала функция `getInt` конвертирует каждый строковый объект в целое число (строка 24), прибавляет к нему 100 (строка 25), снова конвертирует в строковый объект (строка 26) и, наконец, выводит содержимое строкового объекта на консоль (строка 27). Если первоначальная конвертация в тип `int` не удастся, то возвращается строковый литерал `arg + ": Error"` (строка 14) и выводится на консоль.

3. Интерфейс всех контейнеров



Циппи подготавливает пакеты

Краткий обзор

Контейнер – это вместилище, в котором могут храниться объекты других типов. Контейнеры последовательностей и ассоциативные контейнеры стандартной библиотеки шаблонов реализованы как вспомогательные шаблонные классы и имеют много общего. Например, операции создания или удаления контейнера, определения его размера, доступа к его элементам, присваивания или замены не зависят от типа элементов контейнера. У каждого контейнера есть как минимум один параметр типа и выделитель памяти под этот тип. Выделитель памяти (аллокатор) большую часть времени работает в фоновом режиме. Примером может служить контейнер `std::vector`. Вызов `std::vector<int>` приводит к вызову `std::vector<int, std::allocator<int>>`. Благодаря выделителю памяти `std::allocator` можно динамически изменять размеры всех контейнеров, за исключением контейнера `std::array`. Однако у них есть и другие общие черты. К элементам контейнера можно обращаться с помощью итератора.

Несмотря на большое сходство контейнеров, они различаются в деталях. Подробнее об этом говорится в главах «Контейнер последовательности» и «Ассоциативный контейнер».

Имея контейнеры последовательностей `std::array`, `std::vector`, `std::deque`, `std::list` и `std::forward_list`, язык C++ располагает профильным инструментом для каждой области применения.

Ассоциативные контейнеры можно классифицировать на упорядоченные и неупорядоченные.

Создание и удаление

Для каждого контейнера предусмотрены различные конструкторы. Для удаления всех элементов контейнера `cont` можно использовать функцию-член `cont.clear()`. Нет разницы, создается либо удаляется контейнер, добавляются либо удаляются из него элементы – контейнер всякий раз берет на себя управление памятью.

В приведенной ниже таблице показаны конструкторы и деструкторы контейнера. В ней контейнер `std::vector` часто обозначает все остальные.

Таблица. Создание и удаление контейнера

Пример	Тип
<code>std::vector<int> vec1</code>	Конструирование по умолчанию
<code>std::vector<int> vec2(vec1.begin(), vec1.end())</code>	Диапазон
<code>std::vector<int> vec3(vec2)</code>	Копирование
<code>std::vector<int> vec3= vec2</code>	Копирование
<code>std::vector<int> vec4(std::move(vec3))</code>	Перемещение
<code>std::vector<int> vec4= std::move(vec3)</code>	Перемещение
<code>std::vector<int> vec5 {1, 2, 3, 4, 5}</code>	Последовательность (список инициализации)
<code>std::vector<int> vec5= {1, 2, 3, 4, 5}</code>	Последовательность (список инициализации)
<code>vec5.~vector()</code>	Деструктор
<code>vec5.clear()</code>	Удаление элементов

Для экземпляра вспомогательного шаблонного класса `std::array` нужно указывать размер, используемый во время компиляции, и использовать агрегатную инициализацию¹ как метод инициализации агрегатного типа. В классе `std::array` нет функций-членов, которые служат для удаления его элементов.

В следующем ниже примере используются различные конструкторы для различных контейнеров.

¹ См. https://en.cppreference.com/w/cpp/language/aggregate_initialization.

Различные конструкторы

```
// containerConstructor.cpp
...
#include <map>
#include <unordered_map>
#include <vector>

...

using namespace std;

vector<int> vec= {1, 2, 3, 4, 5, 6, 7, 8, 9};
map<string, int> m= {"bart", 12345}, {"jenne", 34929}, {"huber", 840284} };
unordered_map<string, int> um{m.begin(), m.end()};

for (auto v: vec) cout << v << " "; // 1 2 3 4 5 6 7 8 9
for (auto p: m) cout << p.first << "," << p.second << " ";
// bart,12345 huber,840284 jenne,34929

for (auto p: um) cout << p.first << "," << p.second << " ";
// bart,12345 jenne,34929 huber,840284

vector<int> vec2= vec;
cout << vec.size() << endl; // 9
cout << vec2.size() << endl; // 9

vector<int> vec3= move(vec);
cout << vec.size() << endl; // 0
cout << vec3.size() << endl; // 9

vec3.clear();
cout << vec3.size() << endl; // 0
```

Размер контейнеров

С помощью функции-члена `cont.empty()` можно проверять, является контейнер `cont` пустым или нет. Функция-член `cont.size()` возвращает текущее количество элементов, а функция-член `cont.max_size()` – максимальное количество элементов, которое контейнер `cont` может иметь. Максимальное количество элементов определяется реализацией.

Размер контейнера

```
// containerSize.cpp
...
```

```

#include <map>
#include <set>
#include <vector>
...
using namespace std;

vector<int> intVec{1, 2, 3, 4, 5, 6, 7, 8, 9};
map<string, int> str2Int = {"bart", 12345},
                        {"jenne", 34929}, {"huber", 840284}};

set<double> douSet{3.14, 2.5};
cout << intVec.empty() << endl; // false
cout << str2Int.empty() << endl; // false
cout << douSet.empty() << endl; // false

cout << intVec.size() << endl; // 9
cout << str2Int.size() << endl; // 3
cout << douSet.size() << endl; // 2

cout << intVec.max_size() << endl; // 4611686018427387903
cout << str2Int.max_size() << endl; // 384307168202282325
cout << douSet.max_size() << endl; // 461168601842738790

```



Вместо функции-члена `cont.size()` рекомендуется использовать функцию-член `cont.empty()`

Для того чтобы определить, имеет контейнер `cont` элементы или является пустым, вместо `(cont.size() == 0)` рекомендуется использовать функцию-член `cont.empty()`. Во-первых, указанная функция в общем случае работает быстрее, чем `(const.size() == 0)`; во-вторых, вспомогательный шаблонный класс `std::forward_list` не имеет функции-члена `size()`.

Доступ к элементам контейнера

Итератор позволяет обращаться к элементам контейнера. При использовании начального и конечного итераторов получается диапазон, который можно обрабатывать дальше. Применение функции-члена `cont.begin()` к контейнеру `cont` порождает начальный итератор `begin`, а применение функции-члена `cont.end()` – конечной итератор `end`, которые задают полуоткрытый диапазон. Причина, по которой он называется полуоткрытым, связана с тем, что начальный итератор `begin` принадлежит диапазону, а конечной итератор `end` ссылается на позицию, находящуюся сразу за последним элементом диапазона. Указанные два итератора, `cont.begin()` и `cont.end()`, дают возможность модифицировать элементы контейнера.

Таблица. Создание и удаление контейнера

Итератор	Описание
<code>cont.begin()</code> и <code>cont.end()</code>	Пара итераторов для выполнения итераций в прямом направлении
<code>cont.cbegin()</code> и <code>cont.cend()</code>	Пара константных итераторов для выполнения итераций в прямом направлении
<code>cont.rbegin()</code> и <code>cont.rend()</code>	Пара итераторов для выполнения итераций в обратном направлении
<code>cont.crbegin()</code> и <code>cont.crend()</code>	Пара константных итераторов для выполнения итераций в обратном направлении

Теперь элементы контейнера можно модифицировать.

Доступ к элементам контейнера

```
// containerAccess.cpp
...
#include <vector>
...

struct MyInt{
    MyInt(int i): myInt(i){};
    int myInt;
};

std::vector<MyInt> myIntVec;
myIntVec.push_back(MyInt(5));
myIntVec.emplace_back(1);
std::cout << myIntVec.size() << '\n';           // 2

std::vector<int> intVec;
intVec.assign({1, 2, 3});
for (auto v: intVec) std::cout << v << " ";    // 1 2 3

intVec.insert(intVec.begin(), 0);
for (auto v: intVec) std::cout << v << " ";    // 0 1 2 3

intVec.insert(intVec.begin()+4, 4);
for (auto v: intVec) std::cout << v << " ";    // 0 1 2 3 4

intVec.insert(intVec.end(), {5, 6, 7, 8, 9, 10, 11});

for (auto v: intVec) std::cout << v << " ";    // 0 1 2 3 4 5 6 7 8 9 10 11
```

```

for (auto revIt= intVec.rbegin(); revIt != intVec.rend(); ++revIt)
    std::cout << *revIt << " "; // 11 10 9 8 7 6 5 4 3 2 1 0

intVec.pop_back();
for (auto v: intVec ) std::cout << v << " "; // 0 1 2 3 4 5 6 7 8 9 10

```

Присваивание и обмен контейнеров местами

Существующим контейнерам можно присваивать другие контейнеры и менять местами два контейнера. Для присваивания контейнера `cont2` контейнеру `cont` существует копирующее присваивание `cont = cont2` и перемещающее присваивание `cont = std::move(cont2)`. Особой формой присваивания является присваивание со списком инициализации: `cont= {1, 2, 3, 4, 5}`. Для контейнера `std::array` это невозможно. Вспомогательная функция `swap` существует в двух формах: как функция-член `cont.swap(cont2)` и как вспомогательная шаблонная функция `std::swap(cont, cont2)`.

Присваивание и обмен местами

```

// containerAssignmentAndSwap.cpp
...
#include <set>
...

std::set<int> set1{0, 1, 2, 3, 4, 5};
std::set<int> set2{6, 7, 8, 9};

for (auto s: set1) std::cout << s << " "; // 0 1 2 3 4 5
for (auto s: set2) std::cout << s << " "; // 6 7 8 9

set1= set2;
for (auto s: set1) std::cout << s << " "; // 6 7 8 9
for (auto s: set2) std::cout << s << " "; // 6 7 8 9

set1= std::move(set2);
for (auto s: set1) std::cout << s << " "; // 6 7 8 9
for (auto s: set2) std::cout << s << " "; //

set2= {60, 70, 80, 90};
for (auto s: set1) std::cout << s << " "; // 6 7 8 9
for (auto s: set2) std::cout << s << " "; // 60 70 80 90

```

```
std::swap(set1, set2);
for (auto s: set1) std::cout << s << " "; // 60 70 80 90
for (auto s: set2) std::cout << s << " "; // 6 7 8 9
```

Сравнение контейнеров

Контейнеры поддерживают операторы сравнения `==`, `!=`, `<`, `>`, `<=`, `>=`. При сравнении двух контейнеров сравниваются их элементы. При сравнении ассоциативных контейнеров сравниваются их ключи. Неупорядоченные ассоциативные контейнеры поддерживают только операторы сравнения `==` и `!=`.

Сравнение контейнера

```
// containerComparison.cpp
...
#include <array>
#include <set>
#include <unordered_map>
#include <vector>
...
using namespace std;

vector<int> vec1{1, 2, 3, 4};
vector<int> vec2{1, 2, 3, 4};
cout << (vec1 == vec2) << endl; // true

array<int, 4> arr1{1, 2, 3, 4};
array<int, 4> arr2{1, 2, 3, 4};
cout << (arr1 == arr2) << endl; // true

set<int> set1{1, 2, 3, 4};
set<int> set2{4, 3, 2, 1};
cout << (set1 == set2) << endl; // true

set<int> set3{1, 2, 3, 4, 5};
cout << (set1 < set3) << endl; // true

set<int> set4{1, 2, 3, -3};
cout << (set1 > set4) << endl; // true

unordered_map<int, string> uSet1{{1, "one"}, {2, "two"}};
unordered_map<int, string> uSet2{{1, "one"}, {2, "Two"}};
cout << (uSet1 == uSet2) << endl; // false
```

Стирание контейнеров

Вспомогательные шаблонные функции `std::erase(cont, val)` и `std::erase_if(cont, pred)` стирают все элементы контейнера `cont`, равные `val`, либо выполняют предикат `pred`. Обе функции возвращают количество стертых элементов.

Последовательное стирание контейнеров

```
// erase.cpp
...

template <typename Cont>
void eraseVal(Cont& cont, int val) {
    std::erase(cont, val);
}

template <typename Cont, typename Pred>
void erasePredicate(Cont& cont, Pred pred) {
    std::erase_if(cont, pred);
}

template <typename Cont>
void printContainer(Cont& cont) {
    for (auto c: cont) std::cout << c << " ";
    std::cout << '\n';
}

template <typename Cont>
void doAll(Cont& cont) {
    printContainer(cont);
    eraseVal(cont, 5);
    printContainer(cont);
    erasePredicate(cont, [](auto i) { return i >= 3; });
    printContainer(cont);
}

...

std::string str{"A sentence with e."};
std::cout << "str: " << str << '\n';
std::erase(str, 'e');
std::cout << "str: " << str << '\n';

std::cout << "\nstd::vector " << '\n';
std::vector vec{1, 2, 3, 4, 5, 6, 7, 8, 9};
doAll(vec);
```

```
std::cout << "\nstd::deque " << '\n';  
std::deque deq{1, 2, 3, 4, 5, 6, 7, 8, 9};  
doAll(deq);
```

```
std::cout << "\nstd::list" << '\n';  
std::list lst{1, 2, 3, 4, 5, 6, 7, 8, 9};  
doAll(lst);
```

Вспомогательные шаблонные функции `erase` и `erase_if` могут применяться ко всем контейнерам стандартной библиотеки шаблонов и контейнероподобным объектам класса `std::string`.

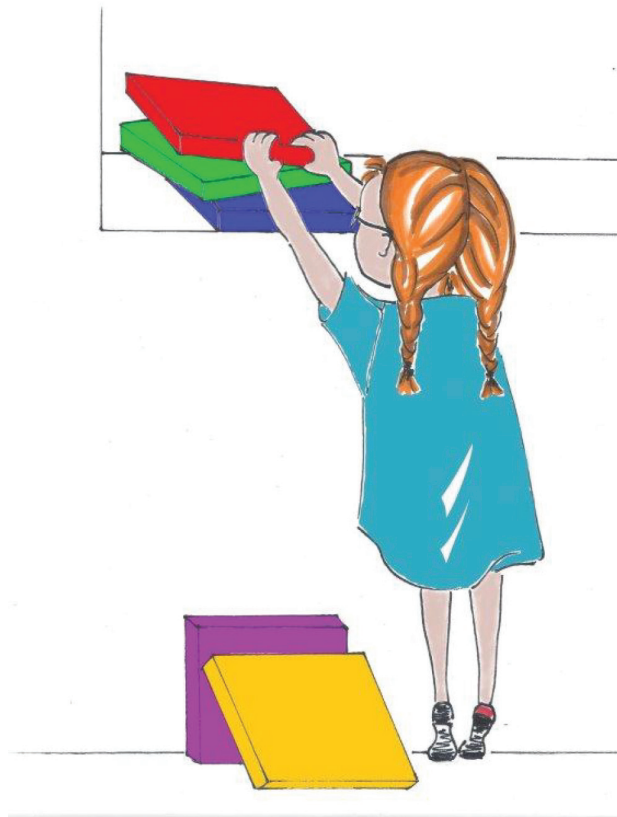
```
str: A sentence with e.  
str: A sntnc with .
```

```
std::vector  
1 2 3 4 5 6 7 8 9  
1 2 3 4 6 7 8 9  
1 2
```

```
std::deque  
1 2 3 4 5 6 7 8 9  
1 2 3 4 6 7 8 9  
1 2
```

```
std::list  
1 2 3 4 5 6 7 8 9  
1 2 3 4 6 7 8 9  
1 2
```

4. Контейнеры последовательностей



Циппи строит стек

Краткий обзор

Контейнеры последовательностей¹ имеют много общего, но у каждого из них есть своя специфическая область применения. Все контейнеры последовательностей реализованы как вспомогательные шаблонные классы. Прежде чем погружаться в детали, ниже представлен обзор всех пяти контейнеров последовательностей именного пространства `std`.

Таблица. Контейнеры последовательностей

Критерий	<code>array</code>	<code>vector</code>	<code>deque</code>	<code>list</code>	<code>forward_list</code>
Размер	Статический	Динамический	Динамический	Динамический	Динамический
Реализация	Статический массив	Динамический массив	Последовательность массивов	Двусвязный список	Односвязный список

¹ См. <http://en.cppreference.com/w/cpp/container>.

Критерий	array	vector	deque	list	forward_list
Доступ	Произвольный	Произвольный	Произвольный	Вперед и назад	Вперед
Оптимизация под вставку и удаление		В конец: $O(1)$	В начало и конец: $O(1)$	В начало и конец: $O(1)$ в произвольную позицию: $O(1)$	В начало $O(1)$ В произвольную позицию: $O(1)$
Резервирование памяти		Да	Нет	Нет	Нет
Высвобождение памяти		shrink_to_fit	shrink_to_fit	Всегда	Всегда
Сила	Нет выделения памяти; минимальные потребности в памяти	95 % решений	Вставка и удаление в начале и в конце	Вставка и удаление в произвольной позиции	Быстрая вставка и удаление; минимальные потребности в памяти
Слабость	Нет динамического выделения памяти	Вставка и удаление в произвольной позиции: $O(n)$	Вставка и удаление в произвольной позиции: $O(n)$	Нет произвольного доступа	Нет произвольного доступа

К приведенной выше таблице необходимо сделать несколько дополнительных замечаний.

$O(i)$ означает сложность операции (время выполнения). $O(1)$ означает, что время выполнения операции на контейнере является постоянным и не зависит от его размера. И наоборот, $O(n)$ означает, что время выполнения линейно зависит от количества элементов в контейнере. Что это значит для контейнера `std::vector`? Время доступа к элементу не зависит от размера контейнера `std::vector`, но вставка или удаление произвольного элемента с количеством элементов, равным k , происходит в k раз медленнее.

Хотя случайный доступ к элементам контейнера `std::vector` имеет ту же сложность $O(1)$, что и случайный доступ к элементам контейнера `std::deque`, это не означает, что обе операции имеют одинаковое быстродействие.

Гарантия сложности $O(1)$ для вставки или удаления в двусвязном списке (контейнер `std::list`) или односвязном списке (контейнер `std::forward_list`) гарантируется только в том случае, если итератор указывает на правильный элемент.



Конкретный класс `std::string` подобен конкретному классу `std::vector<char>`

Разумеется, конкретный класс `std::string` не представлен в виде контейнера стандартной библиотеки шаблонов. Однако с точки зрения поведения его представление похоже на контейнер последовательности, в особенности на конкретный класс `std::vector<char>`. По этой причине строковый объект класса `std::string` будет трактоваться как контейнер класса `std::vector<char>`.

Массивы

1	2	3	4	5	6	7	8	9	10
---	---	---	---	---	---	---	---	---	----

Контейнер `std::array`¹ сочетает в себе характеристики памяти и времени исполнения массива C с интерфейсом контейнера `std::vector`. Контейнер `std::array` представляет собой последовательность фиксированной длины, состоящую из однородных элементов. Это, в частности, означает, что контейнер `std::array` знает свой размер. Для использования контейнеров `std::array` необходим заголовок `<array>`.

При инициализации контейнера `std::array` необходимо следовать нескольким специальным правилам.

```
std::array<int, 10> arr
```

10 элементов не инициализированы.

```
std::array<int, 10> arr{}
```

10 элементов инициализированы по умолчанию.

```
std::array<int, 10> arr{1, 2, 3, 4, 5}
```

Остальные элементы инициализированы по умолчанию.

Класс `std::array` поддерживает три типа доступа к индексам.

```
arr[n];
arr.at(n);
std::get<n>(arr);
```

Наиболее часто используемая первая форма с угловыми скобками не проверяет границы контейнера `arr`. Это противоположно использованию функции-члена `arr.at(n)`. В случае ошибки генерируется исключение `std::range_error`. Последний тип показывает связь контейнера `std::array` с объектом `std::tuple`, так как оба имеют фиксированную длину.

¹ См. <http://en.cppreference.com/w/cpp/container/array>.

Ниже приведено несколько арифметических операций с использованием контейнера `std::array`.

`std::array`

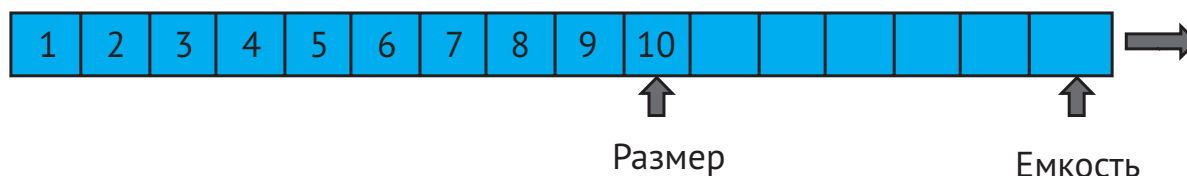
```
// array.cpp
...
#include <array>
...

std::array<int, 10> arr{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
for (auto a: arr) std::cout << a << " "; // 1 2 3 4 5 6 7 8 9 10

double sum= std::accumulate(arr.begin(), arr.end(), 0);
std::cout << sum << '\n';                // 55

double mean= sum / arr.size();
std::cout << mean << '\n';                // 5.5
std::cout << (arr[0] == std::get<0>(arr)); // true
```

Векторы



Контейнер `std::vector`¹ представляет собой последовательность из однородных элементов, длина которого регулируется автоматически во время исполнения. Указанный контейнер нуждается в заголовке `<vector>`. Поскольку контейнер `std::vector` хранит свои элементы в памяти в сплошном порядке, он поддерживает арифметические операции на указателях.

Несколько арифметических операций на указателях

```
for (int i= 0; i < vec.size(); ++i){
    std::cout << vec[i] == *(vec + i) << '\n'; // true
}
```

¹ <http://en.cppreference.com/w/cpp/container/vector>.



При создании контейнера `std::vector` следует различать круглые и фигурные скобки

При создании контейнера `std::vector` необходимо обращать внимание на несколько специальных правил. Конструктор с круглыми скобками в следующем ниже примере создает контейнер `std::vector` с десятью элементами, а конструктор с фигурными скобками – контейнер `std::vector` с элементом 10.

```
std::vector<int> vec(10);
std::vector<int> vec{10};
```

Те же правила соблюдаются и для выражений `std::vector<int>(10, 2011)` и `std::vector<int>{10, 2011}`. В первом случае будет получен контейнер `std::vector` с десятью элементами, инициализированными значением 2011. Во втором случае будет получен контейнер `std::vector` с элементами 10 и 2011. Такое поведение объясняется тем, что фигурные скобки обозначают список инициализации. По этой причине в первом случае используется конструктор последовательности.

Размер в сопоставлении с емкостью

Количество элементов в контейнере `std::vector` обычно меньше, чем количество элементов, для которых уже зарезервировано место. Это происходит по простой причине – размер контейнера `std::vector` может увеличиваться без дорогостоящего выделения новой памяти.

Для разумного использования памяти предназначено несколько операций.

Таблица. Управление памятью контейнера `vec` класса `std::vector`

Функция-член	Описание
<code>vec.size()</code>	Количество элементов в контейнере <code>vec</code>
<code>vec.capacity()</code>	Количество элементов, которое контейнер <code>vec</code> может иметь без перевыделения памяти
<code>vec.resize(n)</code>	Контейнер <code>vec</code> будет увеличен до <code>n</code> элементов
<code>vec.reserve(n)</code>	Резервирование памяти по меньшей мере для <code>n</code> элементов
<code>vec.shrink_to_fit()</code>	Подгонка емкости контейнера <code>vec</code> под его размер

Вызов функции-члена `vec.shrink_to_fit()` не является обязательным. Это значит, что среда исполнения может его игнорировать. Но на популярных платформах всегда наблюдалось желаемое поведение.

Давайте применим эти операции.

```
std::vector
```

```
// vector.cpp
```

```
...
```

```
#include <vector>
```

```

...
std::vector<int> intVec1(5, 2011);
intVec1.reserve(10);
std::cout << intVec1.size() << '\n';    // 5
std::cout << intVec1.capacity() << '\n'; // 10

intVec1.shrink_to_fit();
std::cout << intVec1.capacity() << '\n'; // 5

std::vector<int> intVec2(10);
std::cout << intVec2.size() << '\n';    // 10

std::vector<int> intVec3{10};
std::cout << intVec3.size() << '\n';    // 1

std::vector<int> intVec4{5, 2011};
std::cout << intVec4.size() << '\n';    // 2

```

У контейнера `vec` класса `std::vector` есть несколько функций-членов, которые служат для доступа к своим элементам. С помощью функции-члена `vec.front()` можно получать первый элемент вектора `vec`, а с помощью функции-члена `vec.back()` – последний элемент. Для чтения или записи $(n+1)$ -го элемента контейнера `vec` используется индексный оператор `vec[n]` или функция-член `vec.at(n)`. Последняя проверяет границы контейнера, вследствие чего в случае выхода за пределы контейнера будет сгенерировано исключение `std::out_of_range`.

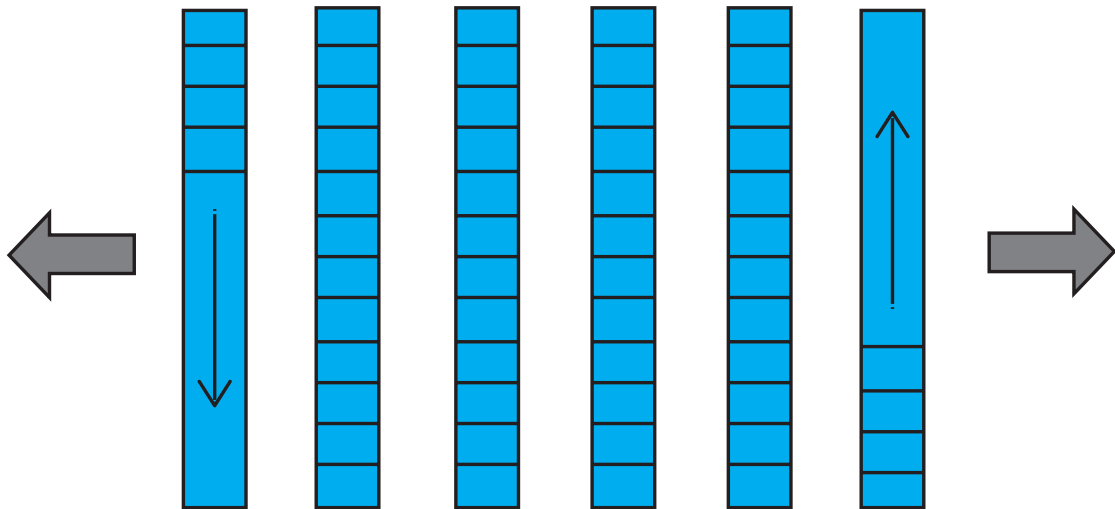
Помимо индексного оператора, контейнер `std::vector` предлагает дополнительные функции-члены, служащие для присваивания, вставки, создания или удаления элементов. Ниже приведен их краткий обзор.

Таблица. Изменение элементов контейнера `vec` класса `std::vector`

Функция-член	Описание
<code>vec.assign(...)</code>	Присваивает один или несколько элементов, диапазон или список инициализации
<code>vec.clear()</code>	Удаляет все элементы из контейнера <code>vec</code>
<code>vec.emplace(pos, args ...)</code>	Создает новый элемент перед позицией <code>pos</code> с аргументами в контейнере <code>vec</code> и возвращает новую позицию элемента
<code>vec.emplace_back(args ...)</code>	Создает новый элемент в контейнере <code>vec</code> с аргументами <code>args ...</code>
<code>vec.erase(...)</code>	Удаляет один элемент или диапазон и возвращает следующую позицию
<code>vec.insert(pos, ...)</code>	Вставляет один или несколько элементов, диапазон или список инициализации и возвращает новую позицию элемента

Функция-член	Описание
<code>vec.pop_back()</code>	Удаляет последний элемент
<code>vec.push_back(elem)</code>	Добавляет копию элемента <code>elem</code> в конец контейнера <code>vec</code>

Очередь с двусторонним доступом



Контейнер `std::deque`¹, который обычно представляет собой последовательность массивов фиксированного размера, очень похож на контейнер `std::vector`. Для указанного класса необходим заголовок `<deque>`. У контейнера `std::deque` есть три дополнительные функции-члена, `deq.push_front(elem)`, `deq.pop_front()` и `deq.emplace_front(args...)`, которые служат для добавления или удаления элементов в его начале.

`std::deque`

```
// deque.cpp
...
#include <deque>
...

struct MyInt{
    MyInt(int i): myInt(i){};
    int myInt;
};

std::deque<MyInt> myIntDeq;

myIntDeq.push_back(MyInt(5));
myIntDeq.emplace_back(1);
```

¹ См. <http://en.cppreference.com/w/cpp/container/deque>.

```

std::cout << myIntDeq.size() << '\n';           // 2

std::deque<MyInt> intDeq;

intDeq.assign({1, 2, 3});
for (auto v: intDeq) std::cout << v << " "; // 1 2 3

intDeq.insert(intDeq.begin(), 0);
for (auto v: intDeq) std::cout << v << " "; // 0 1 2 3

intDeq.insert(intDeq.begin()+4, 4);
for (auto v: intDeq) std::cout << v << " "; // 0 1 2 3 4

intDeq.insert(intDeq.end(), {5, 6, 7, 8, 9, 10, 11});
for (auto v: intDeq) std::cout << v << " "; // 0 1 2 3 4 5 6 7 8 9 10 11

for (auto revIt= intDeq.rbegin(); revIt != intDeq.rend(); ++revIt)
    std::cout << *revIt << " ";           // 11 10 9 8 7 6 5 4 3 2 1 0

intDeq.pop_back();
for (auto v: intDeq) std::cout << v << " "; // 0 1 2 3 4 5 6 7 8 9 10

intDeq.push_front(-1);
for (auto v: intDeq) std::cout << v << " "; // -1 0 1 2 3 4 5 6 7 8 9 10

```

Списки



Контейнер `std::list`¹ представляет собой двусвязный список. Он нуждается в заголовке `<list>`.

Несмотря на схожий с контейнерами `std::vector` и `std::deque` интерфейс, контейнер `std::list` сильно от них отличается. Это связано с его структурой.

Контейнер `std::list` уникален следующими ниже свойствами:

- он не поддерживает произвольный доступ;
- доступ к произвольному элементу происходит медленно, потому что в худшем случае приходится прокручивать весь список;
- добавление или удаление элемента происходит быстро, если итератор указывает на нужное место;
- при добавлении или удалении элемента итератор продолжает оставаться в силе.

¹ См. <http://en.cppreference.com/w/cpp/container/list>.

Ввиду своей уникальной структуры контейнер `std::list` имеет несколько специальных функций-членов.

Таблица. Специальные функции-члены контейнера `lis` класса `std::list`

Функция-член	Описание
<code>lis.merge(c)</code>	Объединяет отсортированный контейнер <code>lis</code> с отсортированным списком <code>c</code> , сохраняя контейнер <code>lis</code> отсортированным
<code>lis.merge(c, op)</code>	Объединяет отсортированный контейнер <code>lis</code> с отсортированным списком <code>c</code> , сохраняя контейнер <code>lis</code> отсортированным. В качестве критерия сортировки используется оператор <code>op</code>
<code>lis.remove(val)</code>	Удаляет из контейнера <code>lis</code> все элементы со значением <code>val</code>
<code>lis.remove_if(pre)</code>	Удаляет из контейнера <code>lis</code> все элементы, удовлетворяющие предикату <code>pre</code>
<code>lis.splice(pos, ...)</code>	Вычленяет элементы в контейнере <code>lis</code> перед позицией <code>pos</code> . Элементы могут быть одиночными элементами, диапазонами или списками
<code>lis.unique()</code>	Удаляет смежный элемент с одинаковым значением
<code>lis.unique(pre)</code>	Удаляет смежные элементы, удовлетворяющие предикату <code>pre</code>

Ниже приведен фрагмент исходного кода с несколькими функциями-членами.

```
std::list
```

```
// list.cpp
...
#include <list>

...

std::list<int> list1{15, 2, 18, 19, 4, 15, 1, 3, 18, 5,
                    4, 7, 17, 9, 16, 8, 6, 6, 17, 1, 2};

list1.sort();
for (auto l: list1) std::cout << l << " ";
// 1 1 2 2 3 4 4 5 6 6 7 8 9 15 15 16 17 17 18 18 19

list1.unique();
for (auto l: list1) std::cout << l << " ";
// 1 2 3 4 5 6 7 8 9 15 16 17 18 19

std::list<int> list2{10, 11, 12, 13, 14};
```

```
list1.splice(std::find(list1.begin(), list1.end(), 15), list2);
for (auto l: list1) std::cout << l << " ";
// 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
```

Впереднаправленные списки



Контейнер `std::forward_list`¹ представляет собой односвязный список, которому необходим заголовок `<forward_list>`. Указанный класс имеет радикально сокращенный интерфейс и оптимизирован под минимальные требования к памяти.

Односвязный список `std::forward_list` имеет много общего с двусвязным списком `std::list`:

- он не поддерживает произвольный доступ;
- доступ к произвольному элементу происходит медленно, потому что в худшем случае приходится прокручивать весь список;
- добавление или удаление элемента происходит быстро, если итератор указывает на нужное место;
- при добавлении или удалении элемента итератор продолжает оставаться в силе;
- операции всегда ссылаются на начало контейнера `std::forward_list` или на позицию после текущего элемента.

Особенность контейнера `std::forward_list`, состоящая в возможности выполнять итерации по его элементам только вперед, оказывает значительное влияние на порядок обработки. Так, итераторы не могут уменьшать значение индекса, поэтому такие операции, как `It--`, на итераторах не поддерживаются. По той же причине у данного контейнера нет обратно направленного итератора. Контейнер `std::forward_list` является единственным контейнером последовательности, который не знает своего размера.



Контейнер `std::forward_list` имеет исключительную область применения

Контейнер `std::forward_list` является заменой односвязных списков. Он оптимизирован под минимальное управление памятью и производительностью, если вставка, извлечение или перемещение элементов затрагивает только смежные элементы. Такая ситуация является типичной для алгоритма сортировки.

¹ См. http://en.cppreference.com/w/cpp/container/forward_list.

Ниже приведены специальные функции-члены контейнера `std::forward_list`.

Таблица. Специальные функции-члены контейнера `forw` класса `std::forward_list`

Функция-член	Описание
<code>forw.before_begin()</code>	Возвращает итератор перед первым элементом
<code>forw.emplace_after(pos, args...)</code>	Создает элемент после позиции <code>pos</code> с аргументами <code>args...</code>
<code>forw.emplace_front(args...)</code>	Создает элемент в начале контейнера <code>forw</code> с аргументами <code>args...</code>
<code>forw.erase_after(pos, ...)</code>	Удаляет из контейнера <code>forw</code> элемент <code>pos</code> или диапазон элементов начиная с позиции <code>pos</code>
<code>forw.insert_after(pos, ...)</code>	Вставляет новые элементы после позиции <code>pos</code> . Эти элементы могут быть одиночными элементами, диапазонами или списками инициализации
<code>forw.merge(c)</code>	Объединяет отсортированный контейнер <code>forw</code> с отсортированным списком <code>c</code> , сохраняя контейнер <code>forw</code> отсортированным
<code>forw.merge(c, op)</code>	Объединяет отсортированный контейнер <code>forw</code> с отсортированным списком <code>c</code> , сохраняя контейнер <code>forw</code> отсортированным. В качестве критерия сортировки используется оператор <code>op</code>
<code>forw.splice_after(pos, ...)</code>	Вычленяет элементы в контейнере <code>forw</code> перед позицией <code>pos</code> . Элементы могут быть одиночными элементами, диапазонами или списками
<code>forw.unique()</code>	Удаляет смежные элементы с одинаковым значением
<code>forw.unique(pre)</code>	Удаляет смежные элементы, удовлетворяющие предикату <code>pre</code>

Давайте посмотрим на уникальные функции-члены контейнера `std::forward_list`.

std::forward_list

```
// forwardList.cpp
...
#include<forward_list>
...

using std::cout;

std::forward_list<int> forw;
std::cout << forw.empty() << '\n';    // true
```

```
forw.push_front(7);
forw.push_front(6);
forw.push_front(5);
forw.push_front(4);
forw.push_front(3);
forw.push_front(2);
forw.push_front(1);
for (auto i: forw) cout << i << " "; // 1 2 3 4 5 6 7

forw.erase_after(forw.before_begin());
cout<< forw.front(); // 2

std::forward_list<int> forw2;
forw2.insert_after(forw2.before_begin(), 1);
forw2.insert_after(++forw2.before_begin(), 2);
forw2.insert_after(++(++forw2.before_begin()), 3);
forw2.push_front(1000);
for (auto i= forw2.cbegin(); i != forw2.cend(); ++i) cout << *i << " ";
// 1000 1 2 3

auto IteratorTo5= std::find(forw.begin(), forw.end(), 5);
forw.splice_after(IteratorTo5, std::move(for2));
for (auto i= forw.cbegin(); i != forw.cend(); ++i) cout << *i << " ";
// 2 3 4 5 1000 1 2 3 6 7

forw.sort();
for (auto i= forw.cbegin(); i != forw.cend(); ++i) cout << *i << " ";
// 1 2 2 3 3 4 5 6 7 1000

forw.reverse();
for (auto i= forw.cbegin(); i != forw.cend(); ++i) cout << *i << " ";
// 1000 7 6 5 4 3 3 2 2 1

forw.unique();
for (auto i= forw.cbegin(); i != forw.cend(); ++i) cout << *i << " ";
// 1000 7 6 5 4 3 2 1
```

5. Ассоциативные контейнеры



Циппи ищет в телефонном справочнике

Краткий обзор

В языке C++ имеется 8 ассоциативных контейнеров¹. Четыре ассоциативных контейнера с сортированными ключами: `std::set`, `std::map`, `std::multiset` и `std::multimap`, а остальные четыре – ассоциативные контейнеры с несортированными ключами: `std::unordered_set`, `std::unordered_map`, `std::unordered_multiset` и `std::unordered_multimap`. Все ассоциативные контейнеры реализованы как вспомогательные шаблонные классы.

Ассоциативные контейнеры – особые, и их особенность заключается в том, что они поддерживают все операции, описанные в главе «Интерфейс всех контейнеров».

Все восемь упорядоченных и неупорядоченных контейнеров объединяет то, что они ассоциируют ключ со значением. Ключ используется для получения значения. При разбивке ассоциативных контейнеров на категории нужно ответить на три простых вопроса:

- отсортированы ли ключи?
- связано ли с ключом значение?
- может ли ключ встречаться более одного раза?

¹ См. <http://en.cppreference.com/w/cpp/container>.

Следующая ниже таблица дает ответы на эти три вопроса. В таблице также дается ответ на четвертый вопрос: каково время доступа к ключу в наилучшем случае?

Таблица. Характеристики ассоциативных контейнеров

Ассоциативный контейнер	Сортированный	Связанное значение	Дополнительные идентичные ключи	Время доступа
<code>std::set</code>	да	нет	нет	логарифмическое
<code>std::unordered_set</code>	нет	нет	нет	постоянное
<code>std::map</code>	да	да	нет	логарифмическое
<code>std::unordered_map</code>	нет	да	нет	постоянное
<code>std::multiset</code>	да	нет	да	логарифмическое
<code>std::unordered_multiset</code>	нет	нет	да	постоянное
<code>std::multimap</code>	да	да	да	логарифмическое
<code>std::unordered_multimap</code>	нет	да	да	постоянное

Начиная со стандарта C++98 в языке C++ существуют упорядоченные ассоциативные контейнеры, а начиная со стандарта C++11 в добавление к ним появились неупорядоченные ассоциативные контейнеры. Обе категории ассоциативных контейнеров имеют очень похожий интерфейс, в силу чего следующий ниже пример исходного кода является идентичным для контейнеров `std::map` и `std::unordered_map`. Точнее говоря, интерфейс контейнера `std::unordered_map` является надмножеством интерфейса контейнера `std::map`. То же самое справедливо и для остальных трех неупорядоченных ассоциативных контейнеров. Таким образом, исходный код относительно легко переносится с упорядоченных контейнеров на неупорядоченные.

Контейнеры можно инициализировать с помощью списка инициализации и добавлять новые элементы с помощью индексного оператора. Для доступа к первому элементу пары ключ-значение `p` используется атрибут `p.first` пары, а для второго элемента – атрибут `p.second` пары, где `p.first` представляет ключ, а `p.second` – связанное с ним значение пары.

`std::map` в сопоставлении с `std::unordered_map`

```
// orderedUnorderedComparison.cpp
...
#include <map>
#include <unordered_map>

// std::map

std::map<std::string, int> m {"Dijkstra", 1972}, {"Scott", 1976}};
m["Ritchie"] = 1983;
```

```

std::cout << m["Ritchie"]; // 1983
for(auto p : m) std::cout << "{" << p.first << "," << p.second << "}";
// {Dijkstra,1972},{Ritchie,1983},{Scott,1976}

m.erase("Scott");
for(auto p : m) std::cout << "{" << p.first << "," << p.second << "}";
// {Dijkstra,1972},{Ritchie,1983}

m.clear();
std::cout << m.size() << '\n'; // 0

// std::unordered_map

std::unordered_map<std::string, int> um {"Dijkstra", 1972}, {"Scott", 1976}};
um["Ritchie"] = 1983;
std::cout << um["Ritchie"]; // 1983
for(auto p : um) std::cout << "{" << p.first << "," << p.second << "}";
// {Ritchie,1983},{Scott,1976},{Dijkstra,1972}
um.erase("Scott");
for(auto p : um) std::cout << "{" << p.first << "," << p.second << "}";
// {Ritchie,1983},{Dijkstra,1972}

um.clear();
std::cout << um.size() << '\n'; // 0

```

Между двумя вариантами исполнения программы есть небольшое различие: Ключи контейнера `std::map` упорядочены, а ключи контейнера `std::unordered_map` не упорядочены. Возникает вопрос: почему в языке C++ существуют такие похожие контейнеры? Как уже было показано в таблице выше, ответ на данный вопрос для языка C++ типичен: производительность. Время доступа к ключам неупорядоченного ассоциативного контейнера является постоянным и, следовательно, не зависит от размера контейнера. Если контейнеры достаточно велики, то разница в производительности становится существенной. Обратитесь к разделу о производительности в этой главе.

Проверка на наличие элемента

Функция-член `associativeContainer.contains(ele)` проверяет наличие элемента `ele` в контейнере `associativeContainer`.

Проверка на наличие элемента в ассоциативном контейнере

```

// containsElement.cpp
...

template <typename AssozCont>
bool containsElement5(const AssozCont& assozCont) {
    return assozCont.contains(5);
}

```

```

}

std::set<int> mySet{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
std::cout << "containsElement5(mySet): "
           << containsElement5(mySet);           // true

std::unordered_set<int> myUnordSet{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
std::cout << "containsElement5(myUnordSet): "
           << containsElement5(myUnordSet); // true

std::map<int, std::string> myMap{ {1, "red"}, {2, "blue"}, {3, "green"} };
std::cout << "containsElement5(myMap): "
           << containsElement5(myMap);           // false

std::unordered_map<int, std::string> myUnordMap{ {1, "red"}, {2, "blue"},
                                                  {3, "green"} };
std::cout << "containsElement5(myUnordMap): "
           << containsElement5(myUnordMap); // false

```

Вставка и удаление

Вставка (функции-члены `insert` и `emplace`) и удаление (функция-член `erase`) элементов в ассоциативных контейнерах аналогичны правилам для контейнера `std::vector`. Для ассоциативного контейнера, который может иметь ключ только один раз, вставка завершается безуспешно, если ключ уже находится в контейнере. Вдобавок упорядоченные ассоциативные контейнеры поддерживают специальную функцию-член `ordAssCont.erase(key)`, которая удаляет все пары с ключом и возвращает их количество.

Вставка и удаление

```

// associativeContainerModify.cpp
...
#include <set>
...

std::multiset<int> mySet{3, 1, 5, 3, 4, 5, 1, 4, 4, 3, 2, 2, 7, 6, 4, 3, 6};

for (auto s: mySet) std::cout << s << " ";
// 1 1 2 2 3 3 3 3 4 4 4 4 5 5 6 6 7

mySet.insert(8);
std::array<int, 5> myArr{10, 11, 12, 13, 14};
mySet.insert(myArr.begin(), myArr.begin()+3);
mySet.insert({22, 21, 20});

```

```
for (auto s: mySet) std::cout << s << " ";
// 1 1 2 2 3 3 3 3 4 4 4 4 5 5 6 6 7 10 11 12 20 21 22

std::cout << mySet.erase(4); // 4
mySet.erase(mySet.lower_bound(5), mySet.upper_bound(15));
for (auto s: mySet) std::cout << s << " ";
// 1 1 2 2 3 3 3 3 20 21 22
```

Упорядоченные ассоциативные контейнеры

Краткий обзор

Упорядоченные ассоциативные контейнеры `std::map` и `std::multimap` связывают свой ключ со значением. Оба они определены в заголовке `<map>`. Упорядоченные ассоциативные контейнеры `std::set` и `std::multiset` нуждаются в заголовке `<set>`. В приведенной выше таблице описаны их подробности.

Все четыре упорядоченных контейнера параметризуются типом, выделителем памяти и функцией сравнения. В зависимости от типа контейнеры имеют дефолтные значения для выделителя и функции сравнения. Объявление контейнеров `std::map` и `std::set` очень наглядно это демонстрирует.

```
template < class key, class val, class Comp= less<key>,
           class Alloc= allocator<pair<const key, val> >
class map;

template < class T, class Comp = less<T>,
           class Alloc = allocator<T> >
class set;
```

Объявление обоих ассоциативных контейнеров показывает, что с контейнером `std::map` ассоциировано значение. Ключ и значение используются для дефолтного выделителя: `allocator<pair<const key, val>>`. Проявив немного фантазии, из выделителя памяти можно извлечь еще больше. Контейнер `std::map` имеет объекты `std::pair<const key, val>`. Ассоциированное значение `val` не играет никакой роли для критерия сортировки: `less<key>`. Все наблюдения справедливы и для контейнеров `std::multimap` и `std::multiset`.

Ключи и значения

Для ключа и значения упорядоченного ассоциативного контейнера существуют особые правила.

- Ключ должен быть сортируемым (по умолчанию `<`), а также копируемым и перемещаемым.

- Значение должно быть конструируемым по умолчанию, а также копируемым и перемещаемым.

Ассоциированный со значением ключ образует пару p , к ключу которой можно обращаться с помощью атрибута $p.first$ пары, а к значению – с помощью атрибута $p.second$ пары.

```
#include <map>
...
std::multimap<char, int> multiMap= {'a', 10}, {'a', 20}, {'b', 30}};
for (auto p: multiMap) std::cout << "{" << p.first << ", " << p.second << "} ";
// {a,10} {a,20} {b,30}
```

Критерий сравнения

По умолчанию критерием сравнения упорядоченных ассоциативных контейнеров является оператор `std::less`. Если в качестве ключа требуется использовать пользовательский тип, то необходимо перегрузить оператор `<`. Для пользовательского типа данных перегрузки оператора `<` будет достаточно, потому что среда исполнения C++ сравнивает два элемента на равенство с помощью отношения `!(elem1<elem2 || elem2<elem1)`.

Критерий сортировки можно указывать в качестве аргумента шаблона. Указанный критерий должен реализовывать строго слабое упорядочивание.



Строго слабое упорядочивание

Строго слабое упорядочивание для критерия сортировки определяется на множестве S , если удовлетворяются следующие ниже требования:

- для элемента s из множества S должно соблюдаться утверждение, что $s < s$ невозможно;
- для всех элементов $s1$ и $s2$ из множества S должно соблюдаться условие, что если $s1 < s2$, то $s2 < s1$ невозможно;
- для всех элементов $s1, s2$ и $s3$, где $s1 < s2$ и $s2 < s3$, должно соблюдаться утверждение, что $s1 < s3$;
- для всех элементов $s1, s2$ и $s3$, для которых $s1$ не сравнимо с $s2$ и $s2$ не сравнимо с $s3$, должно соблюдаться утверждение, что $s1$ не сравнимо с $s3$.

В отличие от определения строго слабого упорядочивания для критерия сортировки, использование критерия сравнения со строго слабым упорядочиванием гораздо проще для контейнера `std::map`.

```
#include <map>
...
std::map<int, std::string, std::greater<int>> int2Str{
    {5, "five"}, {1, "one"}, {4, "four"}, {3, "three"},
    {2, "two"}, {7, "seven"}, {6, "six"} };
```

```
for (auto p: int2Str) std::cout << "{" << p.first << ", " << p.second << "} ";
// {7,seven} {6,six} {5,five} {4,four} {3,three} {2,two} {1,one}
```

Специальные функции поиска

Упорядоченные ассоциативные контейнеры оптимизированы под поиск, поэтому они предлагают уникальные функции поиска.

Таблица. Специальные функции-члены, предназначенные для поиска в упорядоченных ассоциативных контейнерах

Функция поиска	Описание
<code>ordAssCont.count(key)</code>	Возвращает количество значений с ключом <code>key</code>
<code>ordAssCont.find(key)</code>	Возвращает итератор на элемент, содержащий этот ключ <code>key</code> в контейнере <code>ordAssCont</code> . Если ключ <code>key</code> в контейнере <code>ordAssCont</code> не найден, то возвращает концевой итератор <code>ordAssCont.end()</code>
<code>ordAssCont.upper_bound(key)</code>	Возвращает итератор на первую позицию в контейнере <code>ordAssCont</code> , которая строго больше ключа <code>key</code> и куда ключ будет вставлен
<code>ordAssCont.lower_bound(key)</code>	Возвращает итератор на первую позицию в контейнере <code>ordAssCont</code> , которая не меньше ключа <code>key</code> и куда ключ будет вставлен
<code>ordAssCont.equal_range(key)</code>	Возвращает диапазон <code>ordAssCont.lower_bound(key)</code> и <code>ordAssCont.upper_bound(key)</code> в виде объекта класса <code>std::pair</code>

Теперь рассмотрим применение отдельных функций поиска.

Поиск в ассоциативном контейнере

```
// associativeContainerSearch.cpp
...
#include <set>
...

std::multiset<int> mySet{3, 1, 5, 3, 4, 5, 1, 4, 4, 3, 2, 2, 7, 6, 4, 3, 6};

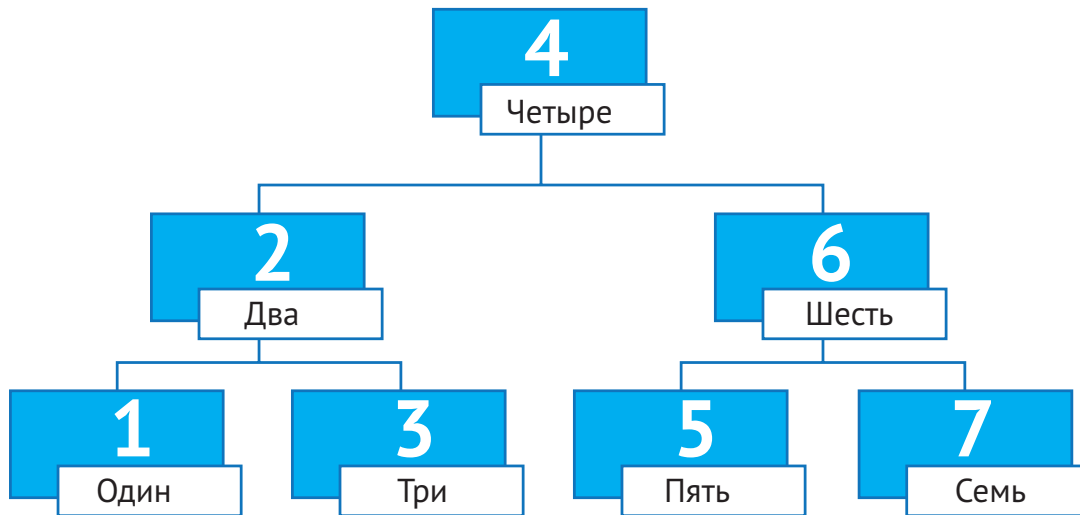
for (auto s: mySet) std::cout << s << " ";
// 1 1 2 2 3 3 3 3 4 4 4 4 5 5 6 6 7

mySet.erase(mySet.lower_bound(4), mySet.upper_bound(4));
for (auto s: mySet) std::cout << s << " ";
// 1 1 2 2 3 3 3 3 5 5 6 6 7

std::cout << mySet.count(3) << '\n'; // 4
```

```
std::cout << *mySet.find(3) << '\n'; // 3
std::cout << *mySet.lower_bound(3) << '\n'; // 3
std::cout << *mySet.upper_bound(3) << '\n'; // 5
auto pair= mySet.equal_range(3);
std::cout << "(" << *pair.first << "," << *pair.second << ")"; // (3,5)
```

std::map



Ассоциативный контейнер `std::map`¹ находит наиболее широкое применение среди программистов на языке C++. Причина проста. Он сочетает в себе зачастую достаточную производительность и очень удобный интерфейс. К его элементам можно обращаться через индексный оператор. Если ключ не существует, то контейнер `std::map` создает пару ключ-значение. Для значения используется дефолтный конструктор.



Контейнер `std::map` следует рассматривать как обобщение контейнера `std::vector`

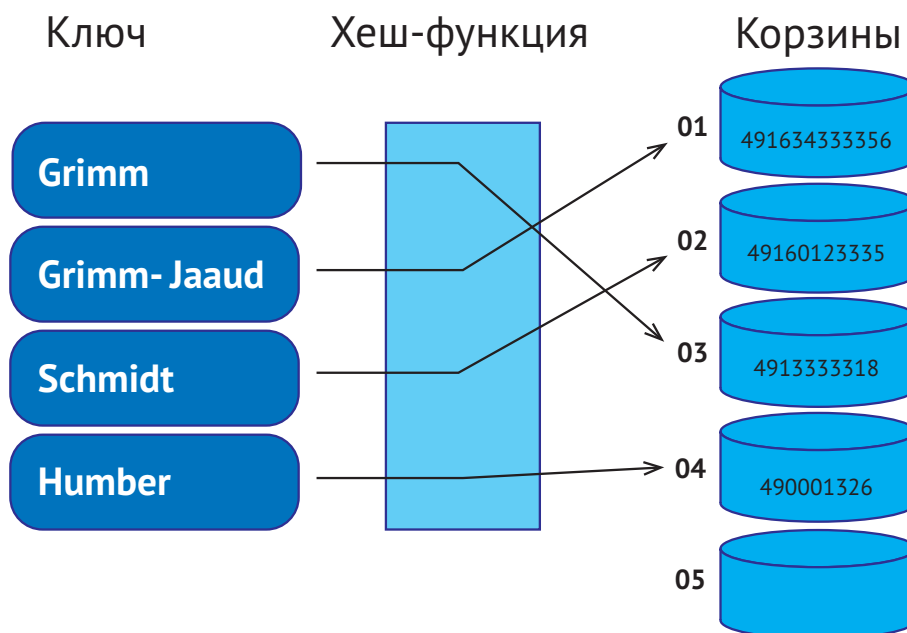
Нередко контейнер `std::map` называют ассоциативным массивом, потому что указанный контейнер поддерживает индексный оператор, как контейнер последовательности. Тонкое отличие кроется в том, что его индекс не ограничен числом, как в случае с контейнером `std::vector`. Его индекс может быть практически любым произвольным типом.

Те же наблюдения справедливы и для его тезки, контейнера `std::unordered_map`.

Дополнительно к индексному оператору контейнер `std::map` поддерживает функцию-член `at`. Доступ посредством указанной функции проверяется. Так, если запрашиваемый ключ в контейнере `std::map` не существует, то генерируется исключение `std::out_of_range`.

¹ См. <http://en.cppreference.com/w/cpp/container/map>.

Неупорядоченные ассоциативные контейнеры



Краткий обзор

В новый стандарт C++11 было введено 4 неупорядоченных ассоциативных контейнера: `std::unordered_map`, `std::unordered_multimap`, `std::unordered_set` и `std::unordered_multiset`. Они имеют много общего со своими тезками – упорядоченными ассоциативными контейнерами. Разница состоит в том, что неупорядоченные ассоциативные контейнеры имеют более богатый интерфейс, а их ключи не сортируются.

Ниже показано объявление неупорядоченного ассоциативного контейнера `std::unordered_map`.

```
template< class key, class val, class Hash= std::hash<key>,
          class KeyEqual= std::equal_to<key>,
          class Alloc= std::allocator<std::pair<const key, val>>>
class unordered_map;
```

Как и контейнер `std::map`, контейнер `std::unordered_map` имеет выделитель памяти, но данный контейнер не нуждается в функции сравнения. Вместо этого ему требуются две дополнительные функциональности: во-первых, вспомогательный шаблонный класс `std::hash<key>`, который обеспечивает средства для вычисления хеш-значения ключа, и, во-вторых, стандартный функциональный объект для сравнения ключей на равенство `std::equal_to<key>`.

Ввиду наличия трех дефолтных параметров шаблона нужно указывать только тип ключа и значение контейнера `std::unordered_map`: `std::unordered_map<char,int> unordMap`.

Ключи и значения

Для ключа и значения неупорядоченного ассоциативного контейнера существуют особые правила:

- ключ должен быть сравнимым на равенство, доступным как хеш-значение и копируемым либо перемещаемым;
- значение должно быть конструируемым по умолчанию и копируемым либо перемещаемым.

Производительность

Производительность – это именно та простая причина, по которой неупорядоченных ассоциативных контейнеров так долго не хватало в языке C++. В приведенном ниже примере один миллион случайно созданных значений читается из 10-миллионных контейнеров `std::map` и `std::unordered_map`. Результат операции чтения впечатляет – линейное время доступа у неупорядоченного ассоциативного контейнера в 20 раз быстрее, чем время доступа у упорядоченного ассоциативного контейнера. Это разница между постоянной и логарифмической сложностью этих операций $O(\log n)$.

Сравнение производительности

```
// associativeContainerPerformance.cpp
...
#include <map>
#include <unordered_map>
...

using std::chrono::duration;
static const long long mapSize= 10000000;
static const long long accSize= 1000000;
...

// чтение 1 миллиона произвольных значений из std::map
// с 10 миллионами значений из randValues
auto start= std::chrono::system_clock::now();
for (long long i= 0; i < accSize; ++i){myMap[randValues[i]];}
duration<double> dur= std::chrono::system_clock::now() - start;
std::cout << dur.count() << " sec";    // 9.18997 sec

// чтение 1 миллиона произвольных значений из std::unordered_map
// с 10 миллионами значений
auto start2= std::chrono::system_clock::now();
for (long long i= 0; i < accSize; ++i){ myUnorderedMap[randValues[i]];}
```

```
duration<double> dur2= std::chrono::system_clock::now() - start2;
std::cout << dur2.count() << " sec"; // 0.411334 sec
```

Хеш-функция

Причиной постоянного времени доступа к неупорядоченному ассоциативному контейнеру является хеш-функция, которая схематично показана на рисунке в начале раздела. Хеш-функция соотносит ключ с его значением, так называемым хеш-значением. Качество хеш-функции является хорошим, если она производит как можно меньше коллизий и равномерно распределяет ключи по корзинам. Поскольку исполнение хеш-функции занимает постоянное количество времени, доступ к элементам в базовом случае также является постоянным.

Хеш-функция:

- уже определена для встроенных типов, таких как булевы величины, натуральные числа и числа с плавающей точкой;
- доступна для контейнероподобных объектов классов `std::string` и `std::wstring`;
- генерирует для последовательности символов `C` типа `const char` хеш-значение адреса указателя;
- может быть определена для пользовательских типов данных.

Говоря о пользовательских типах, используемых в качестве ключа в неупорядоченном ассоциативном контейнере, необходимо помнить о двух требованиях. Они должны иметь хеш-функцию и должны быть сравнимы.

Конкретно-прикладная хеш-функция

```
// unorderedMapHash.cpp
...
#include <unordered_map>
...

struct MyInt{
    MyInt(int v):val(v){}
    bool operator== (const MyInt& other) const {
        return val == other.val;
    }
    int val;
};

struct MyHash{
    std::size_t operator()(MyInt m) const {
```

```

        std::hash<int> hashVal;
        return hashVal(m.val);
    }
};

std::ostream& operator << (std::ostream& st, const MyInt& myIn){
    st << myIn.val ;
    return st;
}

typedef std::unordered_map<MyInt, int, MyHash> MyIntMap;
MyIntMap myMap{{MyInt(-2), -2}, {MyInt(-1), -1}, {MyInt(0), 0}, {MyInt(1), 1}};

for(auto m : myMap) std::cout << "{" << m.first << "," << m.second << "} ";
// {MyInt(1),1} {MyInt(0),0} {MyInt(-1),-1} {MyInt(-2),-2}

std::cout << myMap[MyInt(-2)] << '\n'; // -2

```

Неупорядоченные ассоциативные контейнеры хранят свои индексы в корзинах. Хеш-функция, которая соотносит ключ с индексом, определяет корзину, в которую попадает индекс. Ситуация, когда разные ключи соотносятся с одинаковым индексом, называется коллизией. Хеш-функция старается избегать таких ситуаций.

Индексы обычно хранятся в корзине в виде связного списка. Доступ к корзине является постоянным, а доступ внутри корзины – линейным. Количество корзин называется емкостью. Среднее количество элементов в каждой корзине называется коэффициентом нагрузки. В общем случае среда исполнения C++ генерирует новые корзины, если коэффициент нагрузки превышает 1. Этот процесс называется перехешированием и может инициироваться явным образом:

Подробнее о хеш-функции

```

// hashInfo.cpp
...
#include <unordered_set>

...

using namespace std;

void getInfo(const unordered_set<int>& hash){
    cout << "hash.bucket_count(): " << hash.bucket_count();
    cout << "hash.load_factor(): " << hash.load_factor();
}

```

```
unordered_set<int> hash;
cout << hash.max_load_factor() << endl; // 1

getInfo(hash);
    // hash.bucket_count(): 1
    // hash.load_factor(): 0

hash.insert(500);
cout << hash.bucket(500) << endl;      // 5

// добавить 100 произвольных значений
fillHash(hash, 100);

getInfo(hash);
    // hash.bucket_count(): 109
    // hash.load_factor(): 0.88908

hash.rehash(500);

getInfo(hash);
    // hash.bucket_count(): 541
    // hash.load_factor(): 0.17298
cout << hash.bucket(500);              // 500
```

С помощью функции-члена `max_load_factor` можно читать и устанавливать коэффициент нагрузки. Таким образом, есть возможность влиять на вероятность коллизий и перехеширования. В приведенном выше коротком примере подчеркивается один момент. Ключ 500 сначала находится в 5-й корзине, но после перехеширования оказывается в 500-й корзине.

6. Адаптеры контейнеров



Циппи складывает фигурки в коробку

Краткий обзор

Адаптеры контейнеров в языке C++ представляют собой особый тип контейнера, который изменяет интерфейс базисного типа контейнера. Они предоставляют ограниченный интерфейс, используя при этом существующие типы контейнеров.

Линейные контейнеры

В языке C++ есть три специальных контейнера последовательности: `std::stack`, `std::queue` и `std::priority_queue`. Большинство читателей знают эти классические структуры данных из уроков информатики.

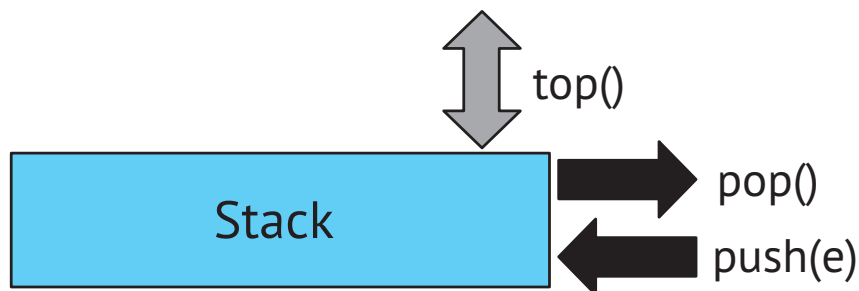
Адаптеры контейнеров:

- поддерживают сокращенный интерфейс у существующих контейнеров последовательностей;
- не могут использоваться с алгоритмами стандартной библиотеки шаблонов;

- представляют собой вспомогательные шаблонные классы, которые параметризуются типом данных и контейнером (`std::vector`, `std::list` и `std::deque`);
- по умолчанию используют контейнер `std::deque` в качестве внутреннего контейнера последовательности:

```
template <typename T, typename Container= deque<T>>
class stack;
```

Стек



Контейнер `std::stack`¹ работает по принципу «последним вошел, первым вышел» (LIFO, от англ. Last In First Out). Контейнер `std::stack`, который нуждается в заголовке `<stack>`, имеет три специальные функции-члена.

С помощью функции-члена `sta.push(e)` можно вставлять новый элемент `e` на вершину стека, с помощью функции-члена `sta.pop()` – удалять его с вершины и с помощью функции-члена `sta.top()` – ссылаться на него. Стек поддерживает операторы сравнения и знает свой размер. Операции на стеке имеют постоянную сложность.

`std::stack`

```
// stack.cpp
...
#include <stack>

...

std::stack<int> myStack;

std::cout << myStack.empty() << '\n'; // true
std::cout << myStack.size() << '\n'; // 0

myStack.push(1);
myStack.push(2);
myStack.push(3);
std::cout << myStack.top() << '\n'; // 3
```

¹ См. <http://en.cppreference.com/w/cpp/container/stack>.

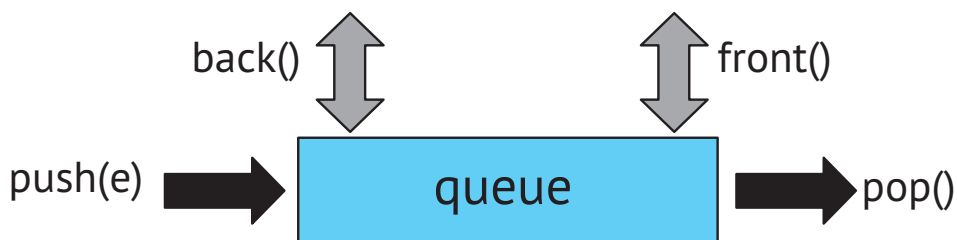
```

while (!myStack.empty()){
    std::cout << myStack.top() << " ";
    myStack.pop();
}                                     // 3 2 1

std::cout << myStack.empty() << '\n'; // true
std::cout << myStack.size() << '\n';  // 0

```

Очередь



Контейнер `std::queue`¹ работает по принципу «первым вошел, первым вышел» (FIFO, от англ. First In First Out). Контейнер `que`, который нуждается в заголовке `<queue>`, имеет четыре специальные функции-члена.

С помощью функции-члена `que.push(e)` можно вставлять элемент `e` в конец очереди и с помощью функции-члена `que.pop()` удалять первый элемент из очереди. Функция-член `que.back()` позволяет обращаться к последнему компоненту очереди, а функция-член `que.front()` – к первому элементу очереди. Контейнер `std::queue` имеет те же характеристики, что и контейнер `std::stack`, в силу чего можно сравнивать экземпляры класса `std::queue` и получать их размеры. Операции на очередях имеют постоянную сложность.

`std::queue`

```

// queue.cpp
...
#include <queue>
...

std::queue<int> myQueue;

std::cout << myQueue.empty() << '\n'; // true
std::cout << myQueue.size() << '\n';  // 0

myQueue.push(1);
myQueue.push(2);

```

¹ См. <http://en.cppreference.com/w/cpp/container/queue>.

```

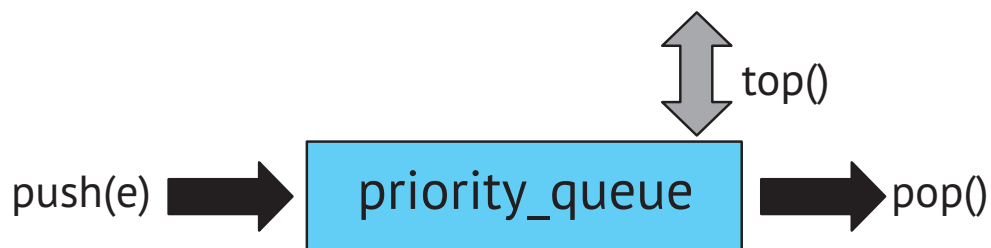
myQueue.push(3);
std::cout << myQueue.back() << '\n'; // 3
std::cout << myQueue.front() << '\n'; // 1

while (!myQueue.empty()){
    std::cout << myQueue.back() << " ";
    std::cout << myQueue.front() << " : ";
    myQueue.pop();
}                                     // 3 1 : 3 2 : 3 3

std::cout << myQueue.empty() << '\n'; // true
std::cout << myQueue.size() << '\n';  // 0

```

Очередь с приоритетом



Контейнер `std::priority_queue`¹ представляет собой сокращенную версию контейнера `std::queue`. Контейнер `std::priority_queue` нуждается в заголовке `<queue>`, и его отличие от контейнера `std::queue` состоит в том, что на вершине очереди с приоритетом всегда находится наибольший элемент. В контейнере `pri` класса `std::priority_queue` по умолчанию используется оператор сравнения `std::less`. Как и в контейнере `std::queue`, функция-член `pri.push(e)` вставляет новый элемент `e` в указанную очередь. Функция-член `pri.pop()` удаляет первый элемент из очереди `pri`, но делает это с логарифмической сложностью. С помощью функции-члена `pri.top()` можно сослаться на первый элемент в очереди, который является самым большим. Контейнер `std::priority_queue` знает свой размер, но не поддерживает оператор сравнения для своих экземпляров.

`std::priority_queue`

```

// priorityQueue.cpp
...
#include <queue>
...
std::priority_queue<int> myPriorityQueue;
std::cout << myPriorityQueue.empty() << '\n'; // true

```

¹ См. http://en.cppreference.com/w/cpp/container/priority_queue.

```
std::cout << myPriorityQueue.size() << '\n'; // 0
myPriorityQueue.push(3);
myPriorityQueue.push(1);
myPriorityQueue.push(2);
std::cout << myPriorityQueue.top() << '\n'; // 3

while (!myPriorityQueue.empty()){
    std::cout << myPriorityQueue.top() << " ";
    myPriorityQueue.pop();
} // 3 2 1

std::cout << myPriorityQueue.empty() << '\n'; // true
std::cout << myPriorityQueue.size() << '\n'; // 0

std::priority_queue<std::string, std::vector<std::string>,
                    std::greater<std::string>> myPriorityQueue2;

myPriorityQueue2.push("Only");
myPriorityQueue2.push("for");
myPriorityQueue2.push("testing");
myPriorityQueue2.push("purpose");
myPriorityQueue2.push(".");

while (!myPriorityQueue2.empty()){
    std::cout << myPriorityQueue2.top() << " ";
    myPriorityQueue2.pop();
} // . Only for purpose testing
```

Ассоциативные контейнеры

Четыре ассоциативных контейнера `std::flat_map`, `std::flat_multimap`, `std::flat_set` и `std::set_multiset` в стандарте C++23 являются упрощенной заменой упорядоченных ассоциативных контейнеров `std::map`, `std::multimap`, `std::set` и `std::multiset`. Точнее говоря, контейнер `std::flat_map` является прямой заменой контейнера `std::map`, контейнер `std::flat_multimap` – прямой заменой контейнера `std::multimap` и т. д.

Плоскоупорядоченные ассоциативные контейнеры нуждаются в отдельных контейнерах последовательностей для своих ключей и значений. Этот контейнер последовательности должен поддерживать итератор с произвольным доступом. По умолчанию в качестве базисного используется контейнер `std::vector`, но также подойдет контейнер `std::array` или `std::deque`.

В следующем ниже фрагменте исходного кода показано объявление контейнеров `std::flat_map` и `std::flat_set`.

```
template<class Key, class T,
        class Compare = less<Key>,
        class KeyContainer = vector<Key>, class MappedContainer = vector<T>>
class flat_map;

template<class Key,
        class Compare = less<Key>,
        class KeyContainer = vector<Key>>
class flat_set;
```

Плоскоупорядоченные ассоциативные контейнеры отличаются по временным и пространственным сложностям от упорядоченных ассоциативных контейнеров. Плоские варианты требуют меньше памяти и быстрее читаются, чем их неплоскоупорядоченные аналоги. В приведенном ниже сравнении плоские и неплоские упорядоченные ассоциативные контейнеры рассматриваются более подробно.



Сравнение плоскоупорядоченного ассоциативного контейнера с их неплоскими аналогами

Плоские варианты обеспечивают более высокую производительность чтения, например обход контейнера, и требуют меньше памяти. Они также нуждаются в том, чтобы элементы были либо копируемыми, либо перемещаемыми. Плоские варианты поддерживают итератор с произвольным доступом.

Неплоские варианты улучшают производительность записи при вставке или удалении элементов. Вдобавок неплоские варианты гарантируют, что итераторы продолжают оставаться в силе после вставки или удаления элементов. Неплоские варианты поддерживают двунаправленные итераторы.

std::sorted_unique

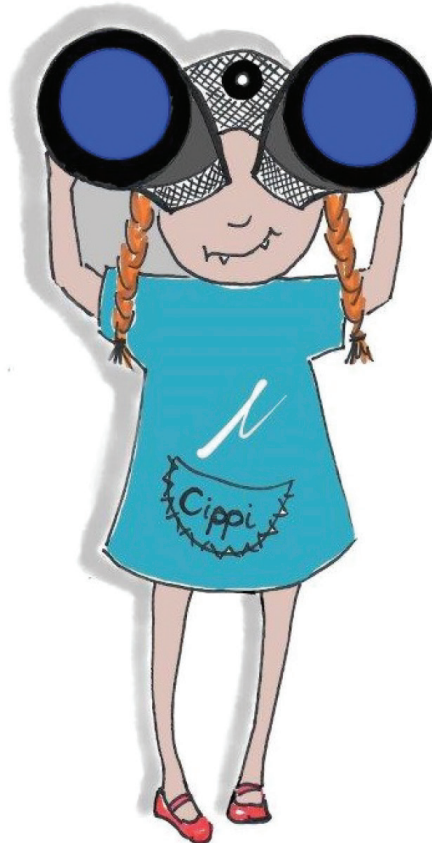
Константа `std::sorted_unique` используется в вызове конструктора или в функциях-членах `insert`, чтобы указывать, что элементы уже отсортированы. Эта мера повышает производительность при создании ассоциативного контейнера с плоским упорядочиванием либо вставке элементов.

Следующий ниже фрагмент исходного кода создает контейнер `std::flat_map` из отсортированного списка инициализации {1, 2, 3, 4, 5}.

```
std::flat_map myFlatMap = { std::sorted_unique, {1, 2, 3, 4, 5},
                                                                {10, 11, 1, 5, -4} };
```

Поведение контейнера при использовании константы `std::sorted_unique` с несортированными элементами не определено.

7. Виды на последовательности объектов



Циппи наблюдает

Краткий обзор

В языке C++ вид (view) – это легковесная абстракция, которая представляет собой ссылку на последовательность смежно расположенных объектов, не владея базисными данными. Виды позволяют работать с данными, обеспечивая возможность выполнять операции на элементах, поддерживая при этом эффективность и безопасность.

Доступ к смежно расположенным объектам

Вспомогательный шаблонный класс `std::span` обозначает объект, который ссылается на последовательность смежно расположенных объектов. Объект шаблонного класса `std::span`, иногда также именуемый видом на последовательность объектов, никогда не является владельцем. Указанная последова-

тельность смежно расположенных объектов может быть обычным массивом `C`, указателем с размером, контейнером `std::array`, контейнером `std::vector` либо контейнером `std::string`. Кроме того, есть возможность обращаться к подпоследовательностям.



Объект класса `std::span` не обесценивается

При вызове функции, которая принимает массив `C`, происходит обесценивание. Функция принимает массив `C` через указатель на его первый элемент. Конвертация массива `C` в указатель чревата ошибками, так как вся информация о длине массива `C` теряется.

В отличие от этого, объект класса `std::span` знает свою длину.

```
// copySpanArray.cpp
...
#include <span>

template <typename T>
void copy_n(const T* p, T* q, int n){}

template <typename T>
void copy(std::span<const T> src, std::span<T> des){}

int arr1[] = {1, 2, 3};
int arr2[] = {3, 4, 5};

copy_n(arr1, arr2, 3);    // (1)
copy<int>(arr1, arr2);    // (2)
```

В отличие от массива `C` (1), функция, которая принимает массив `C` посредством объекта `std::span` (2), не нуждается в явном аргументе длины.

Объект класса `std::span` может иметь статический либо динамический размах. Типичная реализация шаблонного класса `std::span` с динамическим размахом содержит указатель на его первый элемент и длину. По умолчанию объект класса `std::span` имеет динамический размах.

Определение шаблонного класса `std::span`

```
template <typename T, std::size_t Extent = std::dynamic_extent>
class span;
```

В следующей ниже таблице представлены функции-члены класса `std::span`.

Таблица. Функции-члены объекта `std::span` `sp`

Функция-член	Описание
<code>sp.front()</code>	Доступ к первому элементу
<code>sp.back()</code>	Доступ к последнему элементу

Функция-член	Описание
<code>sp[i]</code>	Доступ к <i>i</i> -му элементу
<code>sp.data()</code>	Возвращает указатель на начало последовательности
<code>sp.size()</code>	Возвращает количество элементов последовательности
<code>sp.size_bytes()</code>	Возвращает размер последовательности в байтах
<code>sp.empty()</code>	Возвращает <code>true</code> , если последовательность пуста
<code>sp.first<count>()</code> <code>sp.first(count)</code>	Возвращает подпротяженность, состоящую из первых <code>count</code> элементов последовательности
<code>sp.last<count>()</code> <code>sp.last(count)</code>	Возвращает подпротяженность, состоящую из последних <code>count</code> элементов последовательности.
<code>sp.subspan<first, count>()</code> <code>sp.subspan(first, count)</code>	Возвращает подпротяженность, состоящую из <code>count</code> элементов, начиная с первого

Прием разных аргументов объектом `std::span`

```
// printSpan.cpp
...
#include <span>

void printMe(std::span<int> container) {
    std::cout << "container.size(): " << container.size() << '\n';
    for(auto e : container) std::cout << e << ' ';
    std::cout << "\n\n";
}

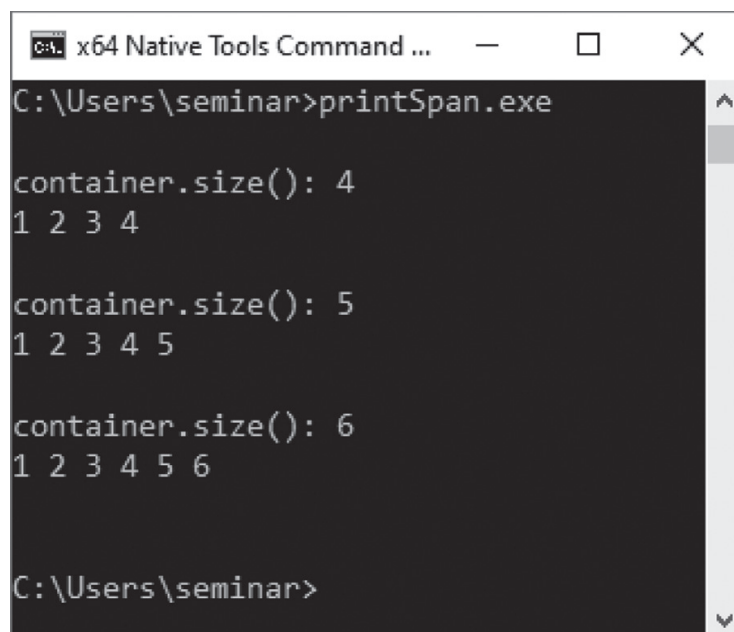
std::cout << '\n';

int arr[]{1, 2, 3, 4};
printMe(arr);                                // (1)

std::vector vec{1, 2, 3, 4, 5};
printMe(vec);                                // (2)

std::array arr2{1, 2, 3, 4, 5, 6};
printMe(arr2);                                // (3)
```

Экземпляр класса `std::span` можно инициализировать массивом `C`, контейнером `std::vector` (1) либо контейнером `std::array` (2).



```

C:\Users\seminar>printSpan.exe

container.size(): 4
1 2 3 4

container.size(): 5
1 2 3 4 5

container.size(): 6
1 2 3 4 5 6

C:\Users\seminar>

```

Автоматическое выведение размера объекта класса `std::span`



Рекомендуется отдавать предпочтение особому конкретному классу `std::string_view` перед классом `std::span`

Конкретный класс `std::string` представляет последовательность смежно расположенных символов. Следовательно, объект класса `std::span` можно инициализировать экземпляром класса `std::string`. Вместе с тем рекомендуется отдавать предпочтение особому конкретному классу `std::string_view`, а не классу `std::span`. Причина в том, что класс `std::string_view` является видом на последовательность символов, а не на последовательность объектов, как `std::span`. Интерфейс класса `std::string_view` похож на интерфейс класса `std::string`, тогда как интерфейс класса `std::span` довольно универсален.

Всю протяженность, а также только ее подпротяженность можно модифицировать. При изменении протяженности модифицируются объекты, на которые имеются ссылки.

Модификация объектов, на которые ссылается объект `std::span`

```

// spanTransform.cpp
...
#include <span>

std::vector vec{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
printMe(vec);

```

```

std::span span1(vec); // (1)
std::span span2{span1.subspan(1, span1.size() - 2)}; // (2)

std::transform(span2.begin(), span2.end(), // (3)
               span2.begin(),
               [](int i){ return i * i; });

printMe(vec);
printMe(span1);

```

В приведенном выше примере `quadSpan.cpp` была определена функция `printMe`. Объект `span1` ссылается на контейнер `std::vector vec` (1). В отличие от него, объект `span2` ссылается только на некоторые элементы внутри контейнера `vec`, исключая первый и последний элементы (2). Следовательно, при отображении каждого элемента на его квадрат происходит обращение только к этим элементам (3).

```

C:\Users\seminar>spanTransform.exe

container.size(): 10
1 2 3 4 5 6 7 8 9 10

container.size(): 10
1 4 9 16 25 36 49 64 81 10

container.size(): 10
1 4 9 16 25 36 49 64 81 10

C:\Users\seminar>

```

Модификация объектов, на которые ссылается объект класса `std::span`

Многомерный доступ

Класс `std::mdspan` представляет собой невладеющий многомерный вид на последовательность смежно расположенных объектов. Нередко такой многомерный вид называют многомерным массивом. Последовательность смежно расположенных объектов может быть обычным массивом C, указателем с размером, контейнером `std::array`, контейнером `std::vector` либо контейнером `std::string`.

Количество размерностей и размер каждой размерности определяют форму многомерного массива. Количество размерностей называется рангом, а размер каждой размерности – размахом. Размер объекта класса `std::mdspan` вычис-

ляется как произведение всех размерностей, которые не равны 0. К элементам объекта `std::mdspan` можно обращаться с помощью многомерного индексного оператора `[]`.

Каждая размерность объекта класса `std::mdspan` может иметь статический либо динамический размах. Статический размах означает, что его длина задается во время компиляции; динамический размах означает, что его длина задается во время исполнения.

Определение шаблонного класса `std::mdspan`

```
template<
    class T,
    class Extents,
    class LayoutPolicy = std::layout_right,
    class AccessorPolicy = std::default_accessor<T>
> class mdspan;
```

- `T`: последовательность смежно расположенных объектов;
- `Extents`: конкретизирует количество размерностей и их размер; каждая размерность может иметь статический либо динамический размах;
- `LayoutPolicy`: конкретизирует политику соотнесения с расположением ячеек памяти с целью обеспечения доступа к базисной памяти;
- `AccessorPolicy`: конкретизирует способ ссылочного обращения к базисным элементам.

Благодаря процедуре выведения аргументов шаблонных классов из типа инициализатора (CTAG, от англ. class template argument deduction)¹ в стандарте C++17 компилятор нередко выводит аргументы шаблона автоматически.

Два двумерных массива

```
// mdspan.cpp

#include <mdspan>
#include <iostream>
#include <vector>

int main() {

    std::vector myVec{1, 2, 3, 4, 5, 6, 7, 8};

    std::mdspan m{myVec.data(), 2, 4};
    std::cout << "m.rank(): " << m.rank() << '\n';
```

¹ См. https://en.cppreference.com/w/cpp/language/class_template_argument_deduction.

```
for (std::size_t i = 0; i < m.extent(0); ++i) {
    for (std::size_t j = 0; j < m.extent(1); ++j) {
        std::cout << m[i, j] << ' ';
    }
    std::cout << '\n';
}

std::cout << '\n';

std::mdspan m2{myVec.data(), 4, 2};
std::cout << "m2.rank(): " << m2.rank() << '\n';

for (std::size_t i = 0; i < m2.extent(0); ++i) {
    for (std::size_t j = 0; j < m2.extent(1); ++j) {
        std::cout << m2[i, j] << ' ';
    }
    std::cout << '\n';
}

}
```

В приведенном выше примере дважды применяется автоматическое выведение аргументов шаблонного класса. В строке 9 оно используется для контейнера `std::vector`, а в строках 11 и 23 – для объекта класса `std::mdspan`. Первый двумерный массив `m` имеет форму (2, 4), второй `m2` – форму (4, 2). В строках 12 и 24 показаны ранги обоих объектов `std::mdspan`. Наличие размаха каждой размерности (строки 14 и 15) и индексного оператора в строке 16 позволяет очень легко выполнять итерации по многомерным массивам.

```
m.rank(): 2
1 2 3 4
5 6 7 8

m2.rank(): 2
1 2
3 4
5 6
7 8
```

Два двумерных массива

Если многомерный массив должен иметь статический размах, то необходимо указывать аргументы шаблона.

Явное указание аргументов шаблона у объекта `std::mdspan`

```
// staticDynamicExtent.cpp

#include <mdspan>

...

std::mdspan<int, std::extents<std::size_t, 2, 4>> m{myVec.data()}; // (1)
std::cout << "m.rank(): " << m.rank() << '\n';

for (std::size_t i = 0; i < m.extent(0); ++i) {
    for (std::size_t j = 0; j < m.extent(1); ++j) {
        std::cout << m[i, j] << ' ';
    }
    std::cout << '\n';
}

std::mdspan<int, std::extents<std::size_t, std::dynamic_extent,
                        std::dynamic_extent>> m2{myVec.data(), 4, 2}; // (2)

std::cout << "m2.rank(): " << m2.rank() << '\n';

for (std::size_t i = 0; i < m2.extent(0); ++i) {
    for (std::size_t j = 0; j < m2.extent(1); ++j) {
        std::cout << m2[i, j] << ' ';
    }
    std::cout << '\n';
}
```

Программа `staticDynamicExtent.cpp` основана на предыдущей программе `mdspan.cpp` и выдает тот же результат, лишь с той разницей, что объект `std::mdspan m` (1) имеет статический размах. Для полноты картины объект `std::mdspan m2` (2) имеет динамический размах. Следовательно, форма объекта `m` задается аргументами шаблона, а форма объекта `m2` – аргументами функции.

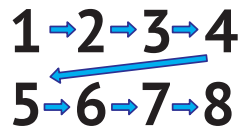
Вспомогательный шаблонный класс `std::mdspan` позволяет конкретизировать политику соотнесения объектов со схемой расположения ячеек памяти с целью организации доступа к базисной памяти. По умолчанию при осуществлении доступа к многомерным данным в объекте `std::mdspan` используется политика построчного соотнесения `std::layout_right` (стиль C, C++ или Python¹), но можно указывать и политику постолбцового соотнесения `std::layout_left` (стиль Fortran² или MATLAB³). На следующем ниже рисунке показана последовательность, с которой происходит обращение к элементам объекта `std::mdspan`.

¹ См. [https://en.wikipedia.org/wiki/Python_\(programming_language\)](https://en.wikipedia.org/wiki/Python_(programming_language)).

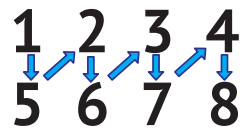
² См. <https://en.wikipedia.org/wiki/Fortran>.

³ См. <https://en.wikipedia.org/wiki/MATLAB>.

std::layout_right



std::layout_left



std::layout_right и std::layout_left

Обход двух объектов std::mdspan с политиками соотнесения std::layout_right и std::layout_left показывает всю разницу.

Использование std::mdspan с политиками соотнесения std::layout_right и std::layout_left

```
// mdspanLayout.cpp
```

```
...
```

```
#include <mdspan>
```

```
std::vector myVec{1, 2, 3, 4, 5, 6, 7, 8};
```

```
std::mdspan<int,                                     // (1)
std::extents<std::size_t, std::dynamic_extent, std::dynamic_extent>,
std::layout_right> m2{myVec.data(), 4, 2};
```

```
std::cout << "m.rank(): " << m.rank() << '\n';
```

```
for (std::size_t i = 0; i < m.extent(0); ++i) {
    for (std::size_t j = 0; j < m.extent(1); ++j) {
        std::cout << m[i, j] << ' ';
    }
    std::cout << '\n';
}
```

```
std::cout << '\n';
```

```
std::mdspan<int,
std::extents<std::size_t, std::dynamic_extent, std::dynamic_extent>,
std::layout_left> m2{myVec.data(), 4, 2};           // (2)
std::cout << "m2.rank(): " << m2.rank() << '\n';
```

```
for (std::size_t i = 0; i < m2.extent(0); ++i) {
    for (std::size_t j = 0; j < m2.extent(1); ++j) {
        std::cout << m2[i, j] << ' ';
    }
    std::cout << '\n';
}
```

В объекте `std::mdspan m` используется политика `std::layout_right` (1), а в другом объекте `std::mdspan m2` – политика `std::layout_left` (2). Благодаря автоматическому выведению аргументов шаблонного класса вызов конструктора `std::mdspan` (1) не требует явных аргументов шаблона и эквивалентен следующему выражению: `std::mdspan m2{myVec.data(), 4, 2}`.

Результат работы программы показывает две разные стратегии соотнесения.

```
m.rank(): 2
1 2
3 4
5 6
7 8

m2.rank(): 2
1 5
2 6
3 7
4 8
```

Объект класса `std::mdspan` с политикой соотнесения `std::layout_left`

В следующей ниже таблице представлен краткий обзор интерфейса вспомогательного шаблонного класса `std::mdspan`.

Таблица. Функции-члены объекта `std::mdspan md`

Функция-член	Описание
<code>md[i]</code>	Доступ к <i>i</i> -му элементу
<code>md.size</code>	Возвращает размер многомерного массива
<code>md.rank</code>	Возвращает размерность многомерного массива
<code>md.extents(i)</code>	Возвращает размер <i>i</i> -й размерности
<code>md.data_handle</code>	Возвращает указатель на последовательность смежно расположенных элементов

8. Итераторы



Циппи делает гигантские шаги

Краткий обзор

С одной стороны, итераторы¹ представляют собой обобщение указателей, которые указывают на позиции в контейнере. С другой, они обеспечивают мощный инструмент выполнения итераций по элементам контейнера и произвольного доступа в контейнере.

Итераторы являются связующим звеном между обобщенными контейнерами и обобщенными алгоритмами стандартной библиотеки шаблонов.

Итераторы поддерживают следующие ниже операции:

- `*` возвращает элемент в текущей позиции;
- `==`, `!=` сравнивает две позиции;
- `=` присваивает новое значение итератору.

Итераторы используются в диапазоне цикла `for` неявным образом.

¹ См. <http://en.cppreference.com/w/cpp/header/iterator>.

Поскольку итераторы не проверяются, они имеют те же проблемы, что и указатели.

```
std::vector<int> vec{1, 23, 3, 3, 3, 4, 5};
std::deque<int> deq;

// Начальный итератор больше концевой итератора
std::copy(vec.begin()+2, vec.begin(), deq.begin());

// Целевой контейнер слишком мал
std::copy(vec.begin(), vec.end(), deq.end());
```

Категории итераторов

Итераторы классифицируются на категории по их способностям. Категория итератора зависит от типа используемого контейнера. В языке C++ есть впереднаправленный итератор, двунаправленный итератор, итератор с произвольным доступом и итератор со сплошным доступом. С помощью впереднаправленного итератора итерации по контейнеру выполняются в прямом направлении, с помощью двунаправленного итератора – в обоих направлениях. Итератор со случайным доступом позволяет обращаться к произвольному элементу напрямую. В частности, итератор с произвольным доступом дает возможность выполнять арифметические действия и упорядочивающие сравнения (например, <). Итератор со сплошным доступом – это итератор с произвольным доступом, для которого необходимо, чтобы элементы контейнера хранились в памяти в сплошном порядке.

В приведенной ниже таблице представлены контейнеры и категории их итераторов. Двунаправленный итератор включает в себя функции впереднаправленного итератора. Итератор с произвольным доступом включает в себя функции впереднаправленного и двунаправленного итераторов. It и It2 – это итераторы, а n – натуральное число.

Таблица. Категории итераторов и контейнеры

Категория итератора	Свойства	Контейнеры
Впереднаправленный итератор	++It, It++, *It It == It2, It != It2	std::unordered_set std::unordered_map std::unordered_multiset std::unordered_multimap std::forward_list
Двунаправленный итератор	--It, It--	std::set std::map std::multiset std::multimap std::list

Категория итератора	Свойства	Контейнеры
Итератор с произвольным доступом	<code>It[i]</code> <code>It+= n, It-= n</code> <code>It+n, It-n</code> <code>n+It</code> <code>It-It2</code> <code>It < It2, It <= It2, It > It2</code> <code>It >= It2</code>	<code>std::deque</code>
Итератор со сплошным доступом		<code>std::array</code> <code>std::vector</code> <code>std::string</code>

Итераторы ввода и вывода – это особые впереднаправленные итераторы: они могут читать и писать элемент, на который они указывают, только один раз.

Критерий итератора

Каждый контейнер генерирует свой подходящий итератор по запросу. Например, контейнер `std::unordered_map` генерирует константные и неконстантные впереднаправленные итераторы.

```
std::unordered_map<std::string, int>::iterator unMapIt= unordMap.begin();
std::unordered_map<std::string, int>::iterator unMapIt= unordMap.end();
std::unordered_map<std::string, int>::const_iterator unMapIt= unordMap.cbegin();
std::unordered_map<std::string, int>::const_iterator unMapIt= unordMap.cend();
```

Кроме того, контейнер `std::map` поддерживает обратконаправленные итераторы:

```
std::map<std::string, int>::reverse_iterator mapIt= map.rbegin();
std::map<std::string, int>::reverse_iterator mapIt= map.rend();

std::map<std::string, int>::const_reverse_iterator mapIt= map.crbegin();
std::map<std::string, int>::const_reverse_iterator mapIt= map.crend();
```



При определении итератора рекомендуется использовать спецификатор `auto`

Определение итераторов считается весьма трудоемким. Автоматическое выведение типов с помощью спецификатора `auto` сводит написание исходного кода к минимуму.

```
std::map<std::string, int>::const_reverse_iterator
mapIt= map.crbegin();
auto mapIt2= map.crbegin();
```

Ниже приведен итоговый пример.

Создание итератора

```
// iteratorCreation.cpp
...
using namespace std;

...

map<string, int> myMap{"Rainer", 1966}, {"Beatrix", 1966}, {"Juliette", 1997},
{"Marius", 1999}];

auto endIt= myMap.end();
for (auto mapIt= myMap.begin(); mapIt != endIt; ++mapIt)
    cout << "{" << mapIt->first << ", " << mapIt->second << "}";
    // {Beatrix,1966},{Juliette,1997},{Marius,1999},{Rainer,1966}

vector<int> myVec{1, 2, 3, 4, 5, 6, 7, 8, 9};
vector<int>::const_iterator vecEndIt= myVec.end();
vector<int>::iterator vecIt;
for (vecIt= myVec.begin(); vecIt != vecEndIt; ++vecIt) cout << *vecIt << " ";
    // 1 2 3 4 5 6 7 8 9

vector<int>::const_reverse_iterator vecEndRevIt= myVec.rend();
vector<int>::reverse_iterator revIt;
for (revIt= myVec.rbegin(); revIt != vecEndRevIt; ++revIt) cout << *revIt << " ";
    // 9 8 7 6 5 4 3 2 1
```

Полезные функции

Вспомогательные шаблонные функции `std::begin`, `std::end`, `std::prev`, `std::next`, `std::distance` и `std::advance` значительно упрощают работу с итераторами. Двухнаправленный итератор нужен только для функции `std::prev`. Всем функциям необходим заголовок `<iterator>`. В приведенной ниже таблице приведен краткий обзор этих вспомогательных функций.

Таблица. Вспомогательные функции для работы с итераторами

Функциональный шаблон	Описание
<code>std::cbegin(cont)</code>	Возвращает константный начальный итератор для контейнера <code>cont</code>
<code>std::cend(cont)</code>	Возвращает константный концевой итератор для контейнера <code>cont</code>

Функциональный шаблон	Описание
<code>std::crbegin(cont)</code>	Возвращает обратноподключенный константный начальный итератор для контейнера <code>cont</code>
<code>std::crend(cont)</code>	Возвращает обратноподключенный константный конечный итератор для контейнера <code>cont</code>
<code>std::prev(it)</code>	Возвращает итератор, указывающий на позицию перед итератором <code>it</code>
<code>std::next(it)</code>	Возвращает итератор, указывающий на позицию после итератора <code>it</code>
<code>std::distance(fir, sec)</code>	Возвращает количество элементов между элементами <code>fir</code> и <code>sec</code>
<code>std::advance(it, n)</code>	Помещает итератор <code>it</code> на <code>n</code> позиций дальше

Теперь давайте взглянем на примеры применения вспомогательных функций.

Вспомогательные функции для работы с итераторами

```
// iteratorUtilities.cpp
...
#include <iterator>

...

using std::cout;

std::unordered_map<std::string, int> myMap{{"Rainer", 1966}, {"Beatrix", 1966},
                                           {"Juliette", 1997}, {"Marius", 1999}};

for (auto m: myMap) cout << "{" << m.first << ", " << m.second << "} ";
    // {Juliette,1997},{Marius,1999},{Beatrix,1966},{Rainer,1966}

auto mapItBegin= std::begin(myMap);
cout << mapItBegin->first << " " << mapItBegin->second; // Juliette 1997

auto mapIt= std::next(mapItBegin);
cout << mapIt->first << " " << mapIt->second; // Marius 1999
cout << std::distance(mapItBegin, mapIt);      // 1

std::array<int, 10> myArr{0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
for (auto a: myArr) std::cout << a << " ";    // 0 1 2 3 4 5 6 7 8 9

auto arrItEnd= std::end(myArr);
auto arrIt= std::prev(arrItEnd);
cout << *arrIt << '\n';                      // 9
```

```
std::advance(arrIt, -5);
cout << *arrIt;           // 4
```

Адаптеры итераторов

Адаптеры итераторов позволяют использовать итераторы в режиме вставки или с потоками данных. Они нуждаются в заголовке `<iterator>`.

Итераторные адаптеры вставки

С помощью трех итераторных адаптеров вставки `std::front_inserter`, `std::back_inserter` и `std::inserter` можно вставлять элемент в контейнер соответственно в начале, в конце либо в произвольную позицию. Все три вспомогательные шаблонные функции соотносят свою функциональность с базисными функциями-членами контейнера `cont`. Память для элементов предоставляется автоматически.

Приведенная ниже таблица показывает две части информации: внутренне используемые функции-члены контейнера и применяемые адаптеры итераторов. Обе зависят от типа контейнера.

Таблица. Три итераторного адаптера вставки

Имя	Внутренне используемая функция-член	Контейнер
<code>std::front_inserter(val)</code>	<code>cont.push_front(val)</code>	<code>std::deque</code> <code>std::list</code>
<code>std::back_inserter(val)</code>	<code>cont.push_back(val)</code>	<code>std::vector</code> <code>std::deque</code> <code>std::list</code> <code>std::string</code>
<code>std::inserter(val, pos)</code>	<code>cont.insert(pos, val)</code>	<code>std::vector</code> <code>std::deque</code> <code>std::list</code> <code>std::string</code> <code>std::map</code> <code>std::set</code>

С тремя итераторными адаптерами вставки можно комбинировать алгоритмы стандартной библиотеки шаблонов.

```
#include <iterator>
...
```

```
std::deque<int> deq{5, 6, 7, 10, 11, 12};
std::vector<int> vec{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15};
```

```

std::copy(std::find(vec.begin(), vec.end(), 13),
          vec.end(), std::back_inserter(deq));

for (auto d: deq) std::cout << d << " ";
// 5 6 7 10 11 12 13 14 15

std::copy(std::find(vec.begin(), vec.end(), 8),
          std::find(vec.begin(), vec.end(), 10),
          std::inserter(deq,
                        std::find(deq.begin(), deq.end(), 10)));d
for (auto d: deq) std::cout << d << " ";
// 5 6 7 8 9 10 11 12 13 14 15

std::copy(vec.rbegin()+11, vec.rend(),
          std::front_inserter(deq));
for (auto d: deq) std::cout << d << " ";
// 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

```

Потоковые итераторные адаптеры

Потоковые итераторные адаптеры могут использовать потоки данных в качестве источника либо поглотителя данных. В языке C++ предлагается две вспомогательные шаблонные функции, служащие для создания итераторных адаптеров для потоков ввода данных (`istream`), и две, служащие для создания итераторных адаптеров для потоков вывода данных (`ostream`). Созданные итераторные адаптеры для потоков ввода ведут себя как итераторы извлечения, а итераторные адаптеры для потоков вывода – как итераторы вставки.

Таблица. Четыре потоковых итераторных адаптера

Функция	Описание
<code>std::istream_iterator<T></code>	Создает итератор конца потока данных
<code>std::istream_iterator<T>(istream)</code>	Создает потоковый итератор для потока ввода данных <code>istream</code>
<code>std::ostream_iterator<T>(ostream)</code>	Создает потоковый итератор для потока вывода данных <code>ostream</code>
<code>std::ostream_iterator<T>(ostream, delim)</code>	Создает потоковый итератор для потока вывода данных <code>ostream</code> с разделителем <code>delim</code>

Благодаря потоковому итераторному адаптеру можно непосредственно читать из потока данных либо писать в него.

Следующий ниже фрагмент интерактивной программы читает натуральные числа в бесконечном цикле из потока ввода `std::cin` и помещает их в контейнер `myIntVec`. Если входные данные не являются натуральными числами, то

происходит ошибка в потоке ввода данных. Все числа из контейнера `myIntVec` копируются в поток вывода `std::cout` через символ `:`. Результат работы программы можно увидеть на консоли.

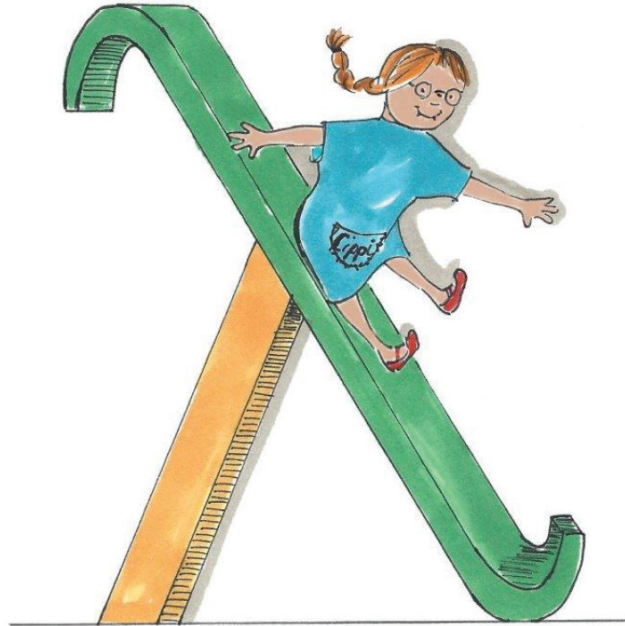
```
#include <iterator>
...

std::vector<int> myIntVec;
std::istream_iterator<int> myIntStreamReader(std::cin);
std::istream_iterator<int> myEndIterator;

// Возможные входные данные
// 1
// 2
// 3
// 4
// z
while(myIntStreamReader != myEndIterator){
    myIntVec.push_back(*myIntStreamReader);
    ++myIntStreamReader;
}

std::copy(myIntVec.begin(), myIntVec.end(),
          std::ostream_iterator<int>(std::cout, ":"));
// 1:2:3:4:
```

9. Вызываемые единицы кода



Циппи катится с горки



В данной главе намеренно опущены исчерпывающие сведения

Данная книга посвящена стандартной библиотеке C++. По этой причине все подробности вызываемых единиц кода здесь рассматриваться не будут. В данной главе будет предоставлено столько информации, сколько необходимо для правильного их использования в алгоритмах стандартной библиотеки шаблонов. Исчерпывающее обсуждение вызываемых единиц кода должно быть частью книги о ядре языка C++.

Краткий обзор

Многие алгоритмы и контейнеры стандартной библиотеки шаблонов могут параметризоваться с помощью вызываемых единиц кода (англ. callable). Вызываемая единица кода ведет себя как функция. Но это не только функция, но и функциональный объект и лямбда-функция. Особыми функциями являются предикаты, которые в качестве результата возвращают булеву величину. Если предикат имеет один аргумент, то он называется унарным предикатом. Если он имеет два аргумента, то предикат называется бинарным. То же самое относится и к функциям. Функция, принимающая один аргумент, является унарной; функция, принимающая два аргумента, – бинарной.



Для того чтобы пользовательский алгоритм мог изменять элементы контейнера, он должен принимать их по ссылке

Вызываемые единицы кода могут получать свои аргументы по значению либо по ссылке из своего контейнера. Для того чтобы они могли изменять элементы контейнера, они должны обращаться к ним напрямую, поэтому вызываемая единица кода должна получать их по ссылке.

Функции

Функции – это простейшие вызываемые единицы кода. У них может отсутствовать какое-либо состояние, кроме статических переменных. Поскольку определение функции зачастую сильно отделено от ее использования или даже находится в другом блоке трансляции, компилятор имеет меньше возможностей по оптимизации результирующего исходного кода.

```
void square(int& i){ i = i*i; }
std::vector<int> myVec{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

std::for_each(myVec.begin(), myVec.end(), square);
for (auto v: myVec) std::cout << v << " "; // 1 4 9 16 25 36 49 64 81 100
```

Функциональные объекты

Прежде всего функциональные объекты не следует называть функторами¹. Термин «функтор» имеет четкое определение и относится к теории категорий.

Функциональные объекты, или объекты-функции², – это экземпляры классов, которые ведут себя как функции. Поскольку функциональные объекты – это объекты, они могут иметь атрибуты и, следовательно, состояние. Это достигается за счет реализации оператора вызова.

```
struct Square{
    void operator()(int& i){i= i*i;}
};

std::vector<int> myVec{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

std::for_each(myVec.begin(), myVec.end(), Square());

for (auto v: myVec) std::cout << v << " "; // 1 4 9 16 25 36 49 64 81 100
```

¹ См. <https://en.wikipedia.org/wiki/Functor>.

² См. <http://en.cppreference.com/w/cpp/utility/functional>.



Прежде чем использовать функциональный объект, сперва необходимо создать экземпляр

На практике нередко можно встретить ошибку, когда в алгоритме используется только имя функционального объекта (`Square`), а не экземпляр функционального объекта (`Square()`): `std::for_each(myVec.begin(), myVec.end(), Square)`. Это, конечно же, ошибка. Нужно использовать экземпляр: `std::for_each(myVec.begin(), myVec.end(), Square())`.

Предопределенные функциональные объекты

В языке C++ предлагается целый ряд предопределенных функциональных объектов. Для них необходим заголовок `<functional>`. Указанные предопределенные функциональные объекты широко применяются для изменения дефолтного поведения контейнеров. Например, ключи упорядоченных ассоциативных контейнеров по умолчанию сортируются с помощью предопределенного функционального объекта `std::less`. Но вместо него может понадобиться функциональный объект `std::greater`:

```
std::map<int, std::string> myDefaultMap; // std::less<int>
std::map<int, std::string, std::greater<int> > mySpecialMap; // std::greater<int>
```

В стандартной библиотеке шаблонов есть функциональные объекты для выполнения арифметических, логических и побитовых операций, а также операций отрицания и сравнения.

Таблица. Предопределенные функциональные объекты

Функциональный объект	Представитель
Отрицание	<code>std::negate<T>()</code>
Арифметические	<code>std::plus<T>()</code> , <code>std::minus<T>()</code> <code>std::multiplies<T>()</code> , <code>std::divides<T>()</code> <code>std::modulus<T>()</code>
Сравнение	<code>std::equal_to<T>()</code> , <code>std::not_equal_to<T>()</code> <code>std::less<T>()</code> , <code>std::greater<T>()</code> <code>std::less_equal<T>()</code> , <code>std::greater_equal<T>()</code>
Логические	<code>std::logical_not<T>()</code> <code>std::logical_and<T>()</code> , <code>std::logical_or<T>()</code>
Побитовые	Bitwise <code>std::bit_and<T>()</code> , <code>std::bit_or<T>()</code> <code>std::bit_xor<T>()</code>

Лямбда-функции

Лямбда-функции, или анонимные функции¹, обеспечивают функциональность на лету, или прямо на месте. Лямбда-функции могут получать свои аргументы по значению либо по ссылке. Они могут захватывать переменные из своей окружающей области видимости по значению, по ссылке, а в стандарте C++14 – по перемещению. Компилятор извлекает много информации и обладает отличным потенциалом оптимизации.

```
std::vector<int> myVec{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
std::for_each(myVec.begin(), myVec.end(), [](int& i){ i = i*i; });
// 1 4 9 16 25 36 49 64 81 100
```

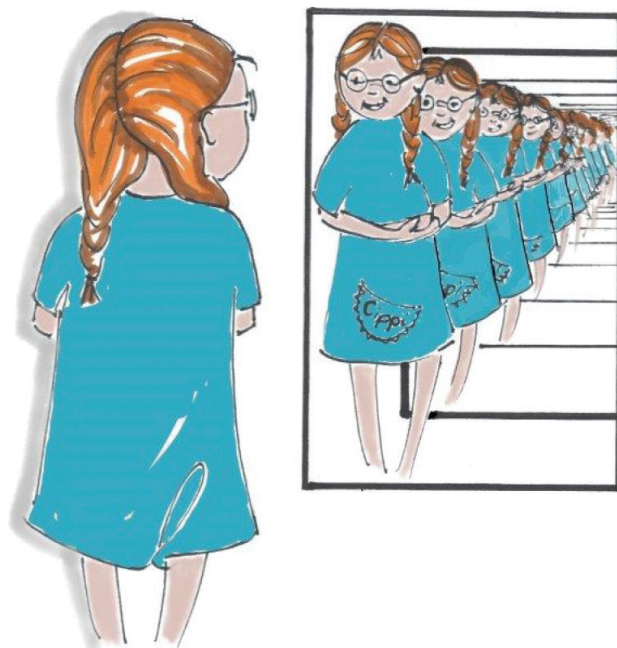


При выборе типа вызываемой единицы кода лямбда-функции должны стоять на первом месте

Если функциональность вызываемой единицы кода коротка и не требует пояснений, то рекомендуется использовать лямбда-функцию. Лямбда-функции, как правило, работают быстрее и более понятны.

¹ См. <http://en.cppreference.com/w/cpp/language/lambda>.

10. Стандартные алгоритмы



Циппи клонирует себя

Краткий обзор

В стандартной библиотеке шаблонов есть раздел стандартных алгоритмов¹. В нем содержится набор вспомогательных шаблонных функций, которые оперируют на контейнерах. Они обеспечивают поддержку сортировки, поиска, манипулирования данными и прочих операций.

Поскольку стандартные алгоритмы являются шаблонными функциями, они не зависят от типа элементов контейнера. Связующим звеном между контейнерами и стандартными алгоритмами являются их итераторы. Если пользовательский контейнер поддерживает интерфейс контейнера стандартной библиотеки шаблонов, то к контейнеру можно применять стандартные алгоритмы.

Обобщенное программирование с помощью стандартных алгоритмов

```
// algorithm.cpp
...
#include <algorithm>
...

template <typename Cont, typename T>
```

¹ См. <http://en.cppreference.com/w/cpp/algorithm>.

```

void doTheSame(Cont cont, T t){
    for (const auto c: cont) std::cout << c << " ";
    std::cout << cont.size() << '\n';
    std::reverse(cont.begin(), cont.end());
    for (const auto c: cont) std::cout << c << " ";
    std::reverse(cont.begin(), cont.end());
    for (const auto c: cont) std::cout << c << " ";
    auto It= std::find(cont.begin(), cont.end(), t);
    std::reverse(It, cont.end());
    for (const auto c: cont) std::cout << c << " ";
}

```

```

std::vector<int> myVec{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
std::deque<std::string> myDeq({"A", "B", "C", "D", "E", "F", "G", "H", "I"});
std::list<char> myList({'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'});

```

```

doTheSame(myVec, 5);
// 1 2 3 4 5 6 7 8 9 10
// 10
// 10 9 8 7 6 5 4 3 2 1
// 1 2 3 4 5 6 7 8 9 10
// 1 2 3 4 10 9 8 7 6 5

```

```

doTheSame(myDeq, "D");
// A B C D E F G H I
// 9
// I H G F E D C B A
// A B C D E F G H I
// A B C I H G F E D

```

```

doTheSame(myList, 'd');
// a b c d e f g h
// 8
// h g f e d c b a
// a b c d e f g h
// a b c h g f e d

```

Общепринятые правила

Для того чтобы использовать стандартные алгоритмы, необходимо придерживаться нескольких правил.

Стандартные алгоритмы определены в различных заголовках. Заголовок `<algorithm>` содержит общие алгоритмы, тогда как заголовок `<numeric>` – численные алгоритмы.

Многие стандартные алгоритмы имеют суффиксы `_if` и `_copy` в имени. Суффикс `_if` обозначает, что алгоритм может параметризовываться предикатом, а суффикс `_copy`, что алгоритм копирует свои элементы в другой диапазон.

Алгоритмы наподобие `auto num = std::count(InpIt first, InpIt last, const T& val)` возвращают количество элементов, равное `val`. Объект `num` имеет тип `iterator_traits<InpIt>::difference_type`. При этом гарантируется, что объекта `num` будет вполне достаточно, чтобы содержать результат. Компилятор будет генерировать верные типы вследствие автоматического вывода возвращаемого типа при использовании спецификатора `auto`.



Если в контейнере используется дополнительный диапазон, то он должен быть допустимым

Стандартный алгоритм `std::copy_if` применяет итератор к началу своего целевого диапазона. Указанный целевой диапазон должен быть допустимым.



Правила именования алгоритмов

В книге используется несколько правил именования типов аргументов и типов значений, возвращаемых из алгоритмов, чтобы облегчить их чтение.

Таблица. Сигнатура алгоритмов

Имя	Описание
<code>InIt</code>	Итератор ввода
<code>FwdIt</code>	Впереднаправленный итератор
<code>BiIt</code>	Двунаправленный итератор
<code>UnFunc</code>	Унарная вызываемая единица кода
<code>BiFunc</code>	Бинарная вызываемая единица кода
<code>UnPre</code>	Унарный предикат
<code>BiPre</code>	Бинарный предикат
<code>Search</code>	Поисковик ¹ включает в свой состав алгоритм поиска
<code>ValType</code>	Из входного диапазона, автоматически выводимый тип значения
<code>Num</code>	<code>typename std::iterator_traits<ForwardIt>::difference_type²</code>
<code>ExePol</code>	Политика исполнения

¹ См. <https://en.cppreference.com/w/cpp/algorithm/search>.

² См. https://en.cppreference.com/w/cpp/iterator/iterator_traits.

Итераторы как связующее звено

Итераторы задают диапазон контейнера, на котором работают алгоритмы. Они описывают полуоткрытый диапазон. В полуоткрытом диапазоне начальный итератор указывает на начало, а конечной итератор – на одну позицию после диапазона.

Итераторы делятся на категории в зависимости от их способностей. См. раздел «Категории» главы «Итераторы». Алгоритмы предоставляют итераторам условия. Как и в случае стандартного алгоритма `std::rotate`, в большинстве случаев достаточно впереднаправленного итератора. Но для стандартного алгоритма `std::reverse` это не подходит. Он нуждается в двунаправленном итераторе.

Последовательное, параллельное либо параллельно-векторизованное исполнение

Используя политику исполнения стандарта C++17, можно конкретизировать режим исполнения стандартного алгоритма, который может быть последовательным, параллельным либо параллельным с векторизацией.



Наличие параллельной стандартной библиотеки шаблонов

По состоянию на 2023 год только в компиляторе Microsoft нативно реализована параллельная версия алгоритмов стандартной библиотеки шаблонов. Для компиляторов GCC и Clang необходимо установить и использовать кросс-платформенную библиотеку Threading Building Blocks¹. Указанная библиотека представляет собой библиотеку шаблонов C++, разработанную в компании Intel, с целью поддержания параллельного программирования на многоядерных процессорах.

Политики исполнения

Тег политики исполнения задает режим, в котором стандартный алгоритм должен работать:

- `std::execution::seq` задает выполнение алгоритма последовательно;
- `std::execution::par`: задает выполнение алгоритма параллельно в нескольких потоках исполнения;
- `std::execution::par_unseq`: задает выполнение алгоритма параллельно в нескольких потоках исполнения и позволяет чередовать отдельные циклы; допускает векторизованную версию с расширениями SIMD² (Single Instruction Multiple Data, т. е. в архитектуре с одной командой и несколькими элементами данных).

¹ См. https://en.wikipedia.org/wiki/Threading_Building_Blocks.

² См. <https://en.wikipedia.org/wiki/SIMD>.

В следующем ниже фрагменте исходного кода показаны все политики исполнения.

Политики исполнения

```
std::vector<int> v = {1, 2, 3, 4, 5, 6, 7, 8, 9};

// стандартная последовательная сортировка
std::sort(v.begin(), v.end());

// последовательное исполнение
std::sort(std::execution::seq, v.begin(), v.end());

// параллельное исполнение
std::sort(std::execution::par, v.begin(), v.end());

// параллельно-векторизованное исполнение
std::sort(std::execution::par_unseq, v.begin(), v.end());
```

Приведенный выше пример показывает, что по-прежнему можно использовать классический вариант сортировки, используя стандартный алгоритм `std::sort`, без политики исполнения. Кроме того, в стандарте C++17 можно конкретизировать используемый вариант в явной форме: последовательный, параллельный либо параллельный с векторизацией.



Параллельно-векторное исполнение

Возможность выполнения алгоритма параллельно-векторизованно обуславливается многочисленными факторами. Например, она зависит от поддержки инструкций SIMD процессором и операционной системой. Кроме того, она зависит от компилятора и уровня оптимизации при компиляции исходного кода. В следующем ниже примере показан простой цикл, в котором создается новый вектор.

```
const int SIZE= 8;

int vec[] = {1, 2, 3, 4, 5, 6, 7, 8};
int res[] = {0, 0, 0, 0, 0, 0, 0, 0};

int main(){
    for (int i = 0; i < SIZE; ++i) {
        res[i] = vec[i] + 5;
    }
}
```

В этом небольшом примере критически важную роль играет строка с выражением `res[i] = vec[i] + 5`. Благодаря Compiler

Explorer¹ можно внимательно изучить инструкции ассемблера, генерируемые компилятором x86-64 clang 3.6.

Без оптимизации

Ниже приведены инструкции ассемблера. Каждое сложение выполняется последовательно.

```
movslq -8(%rbp), %rax
movl   vec(,%rax,4), %ecx
addl   $5, %ecx
movslq -8(%rbp), %rax
movl   %ecx, res(,%rax,4)
```

С максимальной оптимизацией

При применении самого высокого уровня оптимизации -O3 используются специальные регистры, такие как xmm0, которые могут содержать 128 бит или 4 «целых числа», в силу чего сложение происходит параллельно на четырех элементах вектора.

```
movdga -LCPIO_0(%rip), %xmm0 # xmm0 = [5,5,5,5]
movdga vec(%rip), %xmm1
padd   %xmm0, %xmm1
movdga %xmm1, res(%rip)
padd   vec+16(%rip), %xmm0
movdga %xmm0, res+16(%rip)
xorl   %eax, %eax
```

Политикой исполнения могут параметризоваться 77 алгоритмов стандартной библиотеки шаблонов.

Параллельные версии стандартных алгоритмов

Ниже приведен перечень из 77 стандартных алгоритмов, имеющих распараллеленные версии.

Таблица. 77 стандартных алгоритмов с параллельными версиями

std::adjacent_difference	std::adjacent_find	std::all_of
std::any_of	std::copy	std::copy_if
std::copy_n	std::count	std::count_if
std::equal	std::exclusive_scan	std::fill
std::fill_n	std::find	std::find_end
std::find_first_of	std::find_if	std::find_if_not
std::for_each	std::for_each_n	std::generate
std::generate_n	std::includes	std::inclusive_scan
std::inner_product	std::inplace_merge	std::is_heap

¹ См. <https://godbolt.org/>.

<code>std::is_heap_until</code>	<code>std::is_partitioned</code>	<code>std::is_sorted</code>
<code>std::is_sorted_until</code>	<code>std::lexicographical_compare</code>	<code>std::max_element</code>
<code>std::merge</code>	<code>std::min_element</code>	<code>std::minmax_element</code>
<code>std::mismatch</code>	<code>std::move</code>	<code>std::none_of</code>
<code>std::nth_element</code>	<code>std::partial_sort</code>	<code>std::partial_sort_copy</code>
<code>std::partition</code>	<code>std::partition_copy</code>	<code>std::reduce</code>
<code>std::remove</code>	<code>std::remove_copy</code>	<code>std::remove_copy_if</code>
<code>std::remove_if</code>	<code>std::replace</code>	<code>std::replace_copy</code>
<code>std::replace_copy_if</code>	<code>std::replace_if</code>	<code>std::reverse</code>
<code>std::reverse_copy</code>	<code>std::rotate</code>	<code>std::rotate_copy</code>
<code>std::search</code>	<code>std::search_n</code>	<code>std::set_difference</code>
<code>std::set_intersection</code>	<code>std::set_symmetric_difference</code>	<code>std::set_union</code>
<code>std::sort</code>	<code>std::stable_partition</code>	<code>std::stable_sort</code>
<code>std::swap_ranges</code>	<code>std::transform</code>	<code>std::transform_exclusive_scan</code>
<code>std::transform_inclusive_scan</code>	<code>std::transform_reduce</code>	<code>std::uninitialized_copy</code>
<code>std::uninitialized_copy_n</code>	<code>std::uninitialized_fill</code>	<code>std::uninitialized_fill_n</code>
<code>std::unique</code>	<code>std::unique_copy</code>	



Контейнер с ключевым словом `constexpr` и алгоритмы

Стандарт C++20 поддерживает контейнеры `std::vector` и `std::string` с ключевым словом `constexpr`. Ключевое слово `constexpr` означает возможность применять функции-члены обоих контейнеров во время компиляции¹. Вдобавок более 100 алгоритмов² стандартной библиотеки шаблонов объявлены как `constexpr`.

for_each

Стандартный алгоритм `std::for_each` применяет унарную вызываемую единицу кода к каждому элементу своего диапазона. Диапазон задается итераторами ввода.

```
UnFunc std::for_each(InpIt first, InpIt second, UnFunc func)
void std::for_each(ExePol pol, FwdIt first, FwdIt second, UnFunc func)
```

Стандартный алгоритм `std::for_each` ведет себя по-особому при использовании без явно заданной политики исполнения, так как он возвращает свой аргумент, содержащий вызываемую единицу кода. При вызове `std::for_each` с функциональным объектом результат вызова функции можно сохранять непосредственно в функциональном объекте.

¹ Ключевое слово `constexpr` дает возможность создавать исходный код, который выполняется еще до окончания процесса компиляции, что является абсолютным пределом быстродействия программ. – *Прим. перев.*

² См. <https://en.cppreference.com/w/cpp/algorithm>.

```
InpIt std::for_each_n(InpIt first, Size n, UnFunc func)
FwdIt std::for_each_n(ExePol pol, FwdIt first, Size n, UnFunc func)
```

Стандартный алгоритм `std::for_each_n` был введен в стандарт C++17. В нем применяется унарный вызов к первым `n` элементам своего диапазона. Диапазон задается итератором ввода и размером.

`std::for_each`

```
// forEach.cpp
...
#include <algorithm>

...

template <typename T>
class ContInfo{
public:
    void operator()(T t){
        num++;
        sum+= t;
    }
    int getSum() const{ return sum; }
    int getSize() const{ return num; }
    double getMean() const{
        return static_cast<double>(sum)/static_cast<double>(num);
    }
private:
    T sum{0};
    int num{0};
};

std::vector<double> myVec{1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7, 8.8, 9.9};
auto vecInfo= std::for_each(myVec.begin(), myVec.end(), ContInfo<double>());

std::cout << vecInfo.getSum() << '\n'; // 49
std::cout << vecInfo.getSize() << '\n'; // 9
std::cout << vecInfo.getMean() << '\n'; // 5.5

std::array<int, 100> myArr{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
auto arrInfo= std::for_each(myArr.begin(), myArr.end(), ContInfo<int>());

std::cout << arrInfo.getSum() << '\n'; // 55
std::cout << arrInfo.getSize() << '\n'; // 100
std::cout << arrInfo.getMean() << '\n'; // 0.55
```

Немодифицирующие стандартные алгоритмы

Немодифицирующие стандартные алгоритмы – это алгоритмы поиска и подсчета элементов. Помимо них, также есть алгоритмы проверки свойств диапазонов, сравнения диапазонов и поиска диапазонов внутри диапазонов.

Поиск элементов

Поиск элементов можно выполнять тремя разными способами.

Возвращать элемент в диапазоне:

```
InpIt find(InpIt first, InpI last, const T& val)
InpIt find(ExePol pol, FwdIt first, FwdIt last, const T& val)
```

```
InpIt find_if(InpIt first, InpIt last, UnPred pred)
InpIt find_if(ExePol pol, FwdIt first, FwdIt last, UnPred pred)
```

```
InpIt find_if_not(InpIt first, InpIt last, UnPred pre)
InpIt find_if_not(ExePol pol, FwdIt first, FwdIt last, UnPred pre)
```

Возвращать первый элемент в диапазоне:

```
FwdIt1 find_first_of(InpIt1 first1, InpIt1 last1,
                    FwdIt2 first2, FwdIt2 last2)
FwdIt1 find_first_of(ExePol pol, FwdIt1 first1, FwdIt1 last1,
                    FwdIt2 first2, FwdIt2 last2)
```

```
FwdIt1 find_first_of(InpIt1 first1, InpIt1 last1,
                    FwdIt2 first2, FwdIt2 last2, BiPre pre)
FwdIt1 find_first_of(ExePol pol, FwdIt1 first1, FwdIt1 last1,
                    FwdIt2 first2, FwdIt2 last2, BiPre pre)
```

Возвращать одинаковые, смежные элементы в диапазоне:

```
FwdIt adjacent_find(FwdIt first, FwdIt last)
FwdIt adjacent_find(ExePol pol, FwdIt first, FwdIt last)

FwdIt adjacent_find(FwdIt first, FwdI last, BiPre pre)
FwdIt adjacent_find(ExePol pol, FwdIt first, FwdI last, BiPre pre)
```

На входе в алгоритмы в качестве аргументов требуется итератор ввода либо впереднаправленный итератор, а на выходе возвращается итератор на элементе при успешном его отыскании. Если поиск не увенчался успехом, то алгоритмы возвращают концевой итератор.

`std::find`, `std::find_if`, `std::find_if_not`, `std::find_of` и `std::adjacent_fint`

```
// find.cpp
...
```

```

#include <algorithm>

...

using namespace std;

bool isVowel(char c){
    string myVowels{"aeiouäöü"};
    set<char> vowels(myVowels.begin(), myVowels.end());
    return (vowels.find(c) != vowels.end());
}

list<char> myCha{'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j'};
int cha[] = {'A', 'B', 'C'};

cout << *find(myCha.begin(), myCha.end(), 'g');           // g
cout << *find_if(myCha.begin(), myCha.end(), isVowel);     // a
cout << *find_if_not(myCha.begin(), myCha.end(), isVowel); // b

auto iter = find_first_of(myCha.begin(), myCha.end(), cha, cha + 3);
if (iter == myCha.end()) cout << "None of A, B or C."; // None of A, B or C.
auto iter2 = find_first_of(myCha.begin(), myCha.end(), cha, cha + 3,
    [](char a, char b){ return toupper(a) == toupper(b); });

if (iter2 != myCha.end()) cout << *iter2;                 // a
auto iter3 = adjacent_find(myCha.begin(), myCha.end());
if (iter3 == myCha.end()) cout << "No same adjacent chars.";
// No same adjacent chars.
auto iter4 = adjacent_find(myCha.begin(), myCha.end(),
    [](char a, char b){ return isVowel(a) == isVowel(b); });
if (iter4 != myCha.end()) cout << *iter4;                 // b

```

Подсчет элементов

В стандартной библиотеке шаблонов можно подсчитывать элементы с предикатом и без него.

Возвращение количества элементов:

```

Num count(InpIt first, InpIt last, const T& val)
Num count(ExePol pol, FwdIt first, FwdIt last, const T& val)

```

```

Num count_if(InpIt first, InpIt last, UnPred pre)
Num count_if(ExePol pol, FwdIt first, FwdIt last, UnPred pre)

```

Стандартные алгоритмы подсчета на входе в качестве аргументов принимают итераторы ввода и на выходе возвращают количество элементов, соответствующих значению `val` либо предикату.

std::count и std::count_if

```
// count.cpp
...
#include <algorithm>

...

std::string str{"abcdabAAAefaBqeaBCQEaadsfdewAAQAaafbd"};
std::cout << std::count(str.begin(), str.end(), 'a');           // 9
std::cout << std::count_if(str.begin(), str.end(),
                           [](char a){ return std::isupper(a); }); // 12
```

Проверка условий на диапазонах

Три стандартных алгоритма `std::all_of`, `std::any_of` и `std::none_of` отвечают на вопрос, удовлетворяют ли условию все элементы диапазона, хотя бы один из них либо ни один элемент диапазона. В качестве аргументов используются итераторы ввода и унарный предикат, а на выходе возвращается булева величина.

Проверить, все ли элементы диапазона удовлетворяют условию:

```
bool all_of(InpIt first, InpIt last, UnPre pre)
bool all_of(ExePol pol, FwdIt first, FwdIt last, UnPre pre)
```

Проверить, удовлетворяет ли условию хотя бы один элемент диапазона:

```
bool any_of(InpIt first, InpIt last, UnPre pre)
bool any_of(ExePol pol, FwdIt first, FwdIt last, UnPre pre)
```

Проверить, что ни один элемент диапазона не удовлетворяет условию:

```
bool none_of(InpIt first, InpIt last, UnPre pre)
bool none_of(ExePol pol, FwdIt first, FwdIt last, UnPre pre)
```

Как и ранее, ниже приведен пример:

std::all_of, std::any_of и std::none_of

```
// allAnyNone.cpp
...
#include <algorithm>

...

auto even= [](int i){ return i%2; };
std::vector<int> myVec{1, 2, 3, 4, 5, 6, 7, 8, 9};
std::cout << std::any_of(myVec.begin(), myVec.end(), even); // true
```

```
std::cout << std::all_of(myVec.begin(), myVec.end(), even); // false
std::cout << std::none_of(myVec.begin(), myVec.end(), even); // false
```

Сравнение диапазонов

С помощью стандартного алгоритма `std::equal` можно сравнивать диапазоны на равенство. С помощью стандартных алгоритмов `std::lexicographical_compare`, `std::lexicographical_compare_three_way` и `std::mismatch` МОЖНО ВЫЯВЛЯТЬ меньший диапазон.

Выполнить проверку на равенство обоих диапазонов:

```
bool equal(InpIt first1, InpIt last1, InpIt first2)
bool equal(ExePol pol, FwdIt first1, FwdIt last1, FwdIt first2)

bool equal(InpIt first1, InpIt last1, InpIt first2, BiPre pred)
bool equal(ExePol pol, FwdIt first1, FwdIt last1, FwdIt first2, BiPre pred)

bool equal(InpIt first1, InpIt last1,
           InpIt first2, InpIt last2)
bool equal(ExePol pol, FwdIt first1, FwdIt last1,
           FwdIt first2, FwdIt last2)

bool equal(InpIt first1, InpIt last1,
           InpIt first2, InpIt last2, BiPre pred)
bool equal(ExePol pol, FwdIt first1, FwdIt last1,
           FwdIt first2, FwdIt last2, BiPre pred)
```

Проверить, меньше ли первый диапазон, чем второй:

```
bool lexicographical_compare(InpIt first1, InpIt last1,
                           InpIt first2, InpIt last2)
bool lexicographical_compare(ExePol pol, FwdIt first1, FwdIt last1,
                           FwdIt first2, FwdIt last2)

bool lexicographical_compare(InpIt first1, InpIt last1,
                           InpIt first2, InpIt last2, BiPre pred)
bool lexicographical_compare(ExePol pol, FwdIt first1, FwdIt last1,
                           FwdIt first2, FwdIt last2, BiPre pred)
```

Проверить, меньше ли первый диапазон, чем второй. Применяется трехпутное сравнение¹. Возвращается самый сильный тип категории сравнения.

```
bool lexicographical_compare_three_way(InpIt first1, InpIt last1,
                                       InpIt first2, InpIt last2)
bool lexicographical_compare_three_way(InpIt first1, InpIt last1,
                                       InpIt first2, InpIt last2,
                                       BiPre pre)
```

¹ См. https://en.cppreference.com/w/cpp/language/operator_comparison.

Отыскать первую позицию, в которой оба диапазона не равны:

```
pair<InpIt, InpIt> mismatch(InpIt first1, InpIt last1,
                           InpIt first2)
pair<InpIt, InpIt> mismatch(ExePol pol, FwdIt first1, FwdIt last1,
                           FwdIt first2)

pair<InpIt, InpIt> mismatch(InpIt first1, InpIt last1,
                           InpIt first2, BiPre pred)
pair<InpIt, InpIt> mismatch(ExePol pol, FwdIt first1, FwdIt last2,
                           FwdIt first2, BiPre pred)

pair<InpIt, InpIt> mismatch(InpIt first1, InpIt last1,
                           InpIt first2, InpIt last2)
pair<InpIt, InpIt> mismatch(ExePol pol, FwdIt first1, FwdIt last1,
                           FwdIt first2, FwdIt last2)

pair<InpIt, InpIt> mismatch(InpIt first1, InpIt last1,
                           InpIt first2, InpIt last2, BiPre pred)
pair<InpIt, InpIt> mismatch(ExePol pol, FwdIt first1, FwdIt last1,
                           FwdIt first2, FwdIt last2, BiPre pred)
```

Алгоритмы принимают итераторы ввода и в конечном счете бинарный предикат. Алгоритм `std::mismatch` на выходе возвращает пару `pa` итераторов ввода. Член `pa.first` содержит итератор ввода для первого неравного элемента, член `pa.second` содержит соответствующий итератор ввода для второго диапазона. Если оба диапазона являются одинаковыми, то на выходе возвращается два концевых итератора.

`std::equal`, `std::lexicographical_compare` и `std::mismatch`

```
// equalLexicographicalMismatch.cpp
...
#include <algorithm>
...

using namespace std;

string str1{"Only For Testing Purpose." };
string str2{"only for testing purpose." };

cout << equal(str1.begin(), str1.end(), str2.begin()); // false
cout << equal(str1.begin(), str1.end(), str2.begin(),
              [](char c1, char c2){ return toupper(c1) == toupper(c2);} );
// true

str1= {"Only for testing Purpose." };
```

```

str2= {"Only for testing purpose." };

auto pair= mismatch(str1.begin(), str1.end(), str2.begin());
if (pair.first != str1.end()){
    cout << distance(str1.begin(), pair.first)
        << "at (" << *pair.first << ", " << *pair.second << ")"; // 17 at (P,p)
}

auto pair2= mismatch(str1.begin(), str1.end(), str2.begin(),
                    [](char c1, char c2){ return toupper(c1) == toupper(c2); });
if (pair2.first == str1.end()){
    cout << "str1 and str2 are equal"; // str1 and str2 are equal
}

```

Поиск диапазонов внутри диапазонов

Стандартный алгоритм `std::search` ищет диапазон в другом диапазоне с начала диапазона, а алгоритм `std::find_end` – с конца диапазона. Алгоритм `std::search_n` ищет `n` смежно расположенных элементов в диапазоне.

Все алгоритмы принимают впереднаправленный итератор, могут параметризовываться бинарным предикатом и возвращают концевой итератор на первом диапазоне, если поиск не увенчался успехом.

Отыскать второй диапазон в первом и вернуть позицию. Поиск начинается с начала диапазона:

```

FwdIt1 search(FwdIt1 first1, FwdIt1 last1, FwdIt2 first2, FwdIt2 last2)
FwdIt1 search(ExePol pol, FwdIt1 first1, FwdIt1 last1,
              FwdIt2 first2, FwdIt2 last2)

FwdIt1 search(FwdIt1 first1, FwdIt1 last1,
              FwdIt2 first2, FwdIt2 last2, BiPre pre)
FwdIt1 search(ExePol pol, FwdIt1 first1, FwdIt1 last1,
              FwdIt2 first2, FwdIt2 last2, BiPre pre)

FwdIt1 search(FwdIt1 first, FwdIt last1, Search search)

```

Отыскать второй диапазон в первом и вернуть позиции. Поиск начинается с конца диапазона:

```

FwdIt1 find_end(FwdIt1 first1, FwdIt1 last1, FwdIt2 first2 FwdIt2 last2)
FwdIt1 find_end(ExePol pol, FwdIt1 first1, FwdIt1 last1,
                FwdIt2 first2 FwdIt2 last2)

FwdIt1 find_end(FwdIt1 first1, FwdIt1 last1, FwdIt2 first2, FwdIt2 last2,
                BiPre pre)

```


Модифицирующие стандартные алгоритмы

В языке C++ есть целый ряд стандартных алгоритмов, служащих для изменения элементов и диапазонов.

Копирование элементов и диапазонов

Диапазоны можно копировать в прямом направлении с помощью алгоритма `std::copy`, копировать в обратном направлении с помощью алгоритма `std::copy_backward`, а также копировать условно с помощью алгоритма `std::copy_if`. Если нужно скопировать `n` элементов, то можно использовать алгоритм `std::copy_n`.

Скопировать диапазон:

```
OutIt copy(InIt first, InIt last, OutIt result)
FwdIt2 copy(ExePol pol, FwdIt first, FwdIt last, FwdIt2 result)
```

Скопировать `n` элементов:

```
OutIt copy_n(InIt first, Size n, OutIt result)
FwdIt2 copy_n(ExePol pol, FwdIt first, Size n, FwdIt2 result)
```

Скопировать элементы, зависящие от предиката `pre`:

```
OutIt copy_if(InIt first, InIt last, OutIt result, UnPre pre)
FwdIt2 copy_if(ExePol pol, FwdIt first, FwdIt last, FwdIt2 result, UnPre pre)
```

Скопировать диапазон в обратном направлении:

```
BiIt copy_backward(BiIt first, BiIt last, BiIt result)
```

Алгоритмы используют итераторы ввода и копируют свои элементы в диапазон `result`. Они возвращают концевой итератор на целевом диапазоне.

Копирование элементов и диапазонов

```
// copy.cpp
...
#include <algorithm>

...

std::vector<int> myVec{0, 1, 2, 3, 4, 5, 6, 7, 9};
std::vector<int> myVec2(10);

std::copy_if(myVec.begin(), myVec.end(), myVec2.begin()+3,
             [](int a){ return a%2; });

for (auto v: myVec2) std::cout << v << " ";    // 0 0 0 1 3 5 7 9 00

std::string str{"abcdefghijklmnop"};
```

```
std::string str2{"-----"};

std::cout << str2;                // -----
std::copy_backward(str.begin(), str.end(), str2.end());
std::cout << str2;                // ----abcdefghijklmno
std::cout << str;                 // abcdefghijklmnop

std::copy_backward(str.begin(), str.begin() + 5, str.end());
std::cout << str;                 // abcdefghijkabcde
```

Замена элементов и диапазонов

Благодаря стандартным алгоритмам `std::replace`, `std::replace_if`, `std::replace_copy` и `std::replace_copy_if` имеется четыре варианта замены элементов в диапазоне. Алгоритмы различаются по двум аспектам. Во-первых, по наличию у алгоритма предиката. Во-вторых, по копированию элементов алгоритмом в целевой диапазон.

Заменить старые элементы диапазона на новое значение `newValue`, если старый элемент имеет значение `old`.

```
void replace(FwdIt first, FwdIt last, const T& old, const T& newValue)
void replace(ExePol pol, FwdIt first, FwdIt last, const T& old,
             const T& newValue)
```

Заменить старые элементы диапазона на новое значение `newValue`, если старый элемент удовлетворяет предикату `pred`:

```
void replace_if(FwdIt first, FwdIt last, UnPred pred, const T& newValue)
void replace_if(ExePol pol, FwdIt first, FwdIt last, UnPred pred,
               const T& newValue)
```

Заменить старые элементы диапазона на новое значение `newValue`, если старый элемент имеет значение `old` с копированием результата в целевой диапазон `result`:

```
OutIt replace_copy(InpIt first, InpIt last, OutIt result, const T& old,
                  const T& newValue)
FwdIt2 replace_copy(ExePol pol, FwdIt first, FwdIt last,
                   FwdIt2 result, const T& old, const T& newValue)
```

Заменить старые элементы диапазона на новое значение `newValue`, если старый элемент удовлетворяет предикату `pred` с копированием результата в целевой диапазон `result`:

```
OutIt replace_copy_if(InpIt first, InpIt last, OutIt result, UnPred pred,
                    const T& newValue)
FwdIt2 replace_copy_if(ExePol pol, FwdIt first, FwdIt last,
                      FwdIt2 result, UnPred pred, const T& newValue)
```

Алгоритмы в действии:

Замена элементов и диапазонов

```
// replace.cpp
...
#include <algorithm>

...

std::string str{"Only for testing purpose." };
std::replace(str.begin(), str.end(), ' ', '1');
std::cout << str; // Only1for1testing1purpose.

std::replace_if(str.begin(), str.end(), [](char c){ return c == '1'; }, '2');
std::cout << str; // Only2for2testing2purpose.

std::string str2;
std::replace_copy(str.begin(), str.end(), std::back_inserter(str2), '2', '3');
std::cout << str2; // Only3for3testing3purpose.

std::string str3;
std::replace_copy_if(str2.begin(), str2.end(),
std::back_inserter(str3), [](char c){ return c == '3'; }, '4');
std::cout << str3; // Only4for4testing4purpose.
```

Удаление элементов и диапазонов

Четыре вариации стандартных алгоритмов – `std::remove`, `std::remove_if`, `std::remove_copy` и `std::remove_copy_if` – поддерживают два вида операций. С одной стороны, они позволяют удалять элементы из диапазона с предикатом и без него, а с другой – дают возможность копировать результат модификации в новый диапазон.

Удалить из диапазона элементы, имеющие значение `val`:

```
FwdIt remove(FwdIt first, FwdIt last, const T& val)
FwdIt remove(ExePol pol, FwdIt first, FwdIt last, const T& val)
```

Удалить из диапазона элементы, удовлетворяющие предикату `pred`:

```
FwdIt remove_if(FwdIt first, FwdIt last, UnPred pred)
FwdIt remove_if(ExePol pol, FwdIt first, FwdIt last, UnPred pred)
```

Удалить из диапазона элементы, имеющие значение `val`, с копированием результата в новый диапазон `result`:

```
OutIt remove_copy(InpIt first, InpIt last, OutIt result, const T& val)
```

```
FwdIt2 remove_copy(ExePol pol, FwdIt first, FwdIt last,
                   FwdIt2 result, const T& val)
```

Удалить из диапазона элементы, удовлетворяющие предикату `pred`, с копированием результата в новый диапазон `result`.

```
OutIt remove_copy_if(InpIt first, InpIt last, OutIt result, UnPre pred)
FwdIt2 remove_copy_if(ExePol pol, FwdIt first, FwdIt last,
                      FwdIt2 result, UnPre pred)
```

Алгоритмам нужны итератор ввода для исходного диапазона и итератор вывода для целевого диапазона. В качестве результата они возвращают концевой итератор на целевом диапазоне.



При работе с алгоритмами удаления рекомендуется применять идиому «стереть-удалить».

Вариации алгоритмов удаления удаляют элемент из диапазона. Они возвращают новый логический конец диапазона. Использование идиомы «стереть-удалить» предусматривает необходимость корректировать размер контейнера.

Удаление элементов и диапазонов

```
// remove.cpp
...
#include <algorithm>
...

std::vector<int> myVec{0, 1, 2, 3, 4, 5, 6, 7, 8, 9};

auto newIt= std::remove_if(myVec.begin(), myVec.end(),
                          [](int a){ return a%2; });

for (auto v: myVec) std::cout << v << " "; // 0 2 4 6 8 5 6 7 8 9

myVec.erase(newIt, myVec.end());
for (auto v: myVec) std::cout << v << " "; // 0 2 4 6 8

std::string str{"Only for Testing Purpose." };
str.erase( std::remove_if(str.begin(), str.end(),
                          [](char c){ return std::isupper(c); }, str.end() ) );
std::cout << str << '\n'; // nly for esting urpose.
```

Заполнение и создание диапазонов

Заполнение диапазона осуществляется с помощью стандартных алгоритмов `std::fill` и `std::fill_n`, а новые элементы можно генерировать с помощью стандартных алгоритмов `std::generate` и `std::generate_n`.

Заполнить диапазон элементами:

```
void fill(FwdIt first, FwdIt last, const T& val)
void fill(ExePol pol, FwdIt first, FwdIt last, const T& val)
```

Заполнить диапазон `n` новыми элементами:

```
OutIt fill_n(OutIt first, Size n, const T& val)
FwdIt fill_n(ExePol pol, FwdIt first, Size n, const T& val)
```

Сгенерировать диапазон с помощью генератора `gen`:

```
void generate(FwdIt first, FwdIt last, Generator gen)
void generate(ExePol pol, FwdIt first, FwdIt last, Generator gen)
```

Сгенерировать `n` элементов диапазона с помощью генератора `gen`:

```
OutIt generate_n(OutIt first, Size n, Generator gen)
FwdIt generate_n(ExePol pol, FwdIt first, Size n, Generator gen)
```

Алгоритмы ожидают в качестве аргумента значение `val` или генератор `gen`. Генератор `gen` должен быть функцией, не принимающей аргументов и возвращающей новое значение. На выходе из алгоритмов `std::fill_n` и `std::generate_n` находится итератор вывода, указывающий на последний созданный элемент.

Заполнение и создание диапазонов

```
// fillAndCreate.cpp
...
#include <algorithm>

...

int getNext(){
    static int next{0};
    return ++next;
}

std::vector<int> vec(10);
std::fill(vec.begin(), vec.end(), 2011);
for (auto v: vec) std::cout << v << " ";
// 2011 2011 2011 2011 2011 2011 2011 2011 2011 2011

std::generate_n(vec.begin(), 5, getNext);
```

```
for (auto v: vec) std::cout << v << " ";
// 1 2 3 4 5 2011 2011 2011 2011 2011
```

Перемещение диапазонов

Стандартный алгоритм `std::move` перемещает диапазоны вперед, а стандартный алгоритм `std::move_backward` перемещает диапазоны назад.

Переместить диапазон вперед:

```
OutIt move(InpIt first, InpIt last, OutIt result)
FwdIt2 move(ExePol pol, FwdIt first, FwdIt last, Fwd2It result)
```

Переместить диапазон назад:

```
BiIt move_backward(BiIt first, BiIt last, BiIt result)
```

Обоим алгоритмам нужен целевой итератор `result`, к которому перемещается диапазон. В случае алгоритма `std::move` это итератор вывода. В случае алгоритма `std::move_backward` это двунаправленный итератор. Алгоритмы возвращают итератор вывода либо двунаправленный итератор, указывающие на начальную позицию в целевом диапазоне.



Исходный диапазон может быть изменен

Алгоритмы `std::move` и `std::move_backward` применяют семантику перемещения, вследствие чего исходный диапазон остается в силе, но не обязательно имеет те же элементы в последующем.

Перемещение диапазонов

```
// move.cpp
...
#include <algorithm>
...

std::vector<int> myVec{0, 1, 2, 3, 4, 5, 6, 7, 9};
std::vector<int> myVec2(myVec.size());

std::move(myVec.begin(), myVec.end(), myVec2.begin());
for (auto v: myVec2) std::cout << v << " "; // 0 1 2 3 4 5 6 7 9 0

std::string str{"abcdefghijklmno"};
std::string str2{"-----"};
std::move_backward(str.begin(), str.end(), str2.end());
std::cout << str2; // -----abcdefghijklmno
```

Обмен диапазонами

Стандартные алгоритмы `std::swap` и `std::swap_ranges` могут переставлять объекты и диапазоны местами.

Переставить объекты местами:

```
void swap(T& a, T& b)
```

Переставить диапазоны местами:

```
FwdIt swap_ranges(FwdIt1 first1, FwdIt1 last1, FwdIt first2)
FwdIt swap_ranges(ExePol pol, FwdIt1 first1, FwdIt1 last1, FwdIt first2)
```

Возвращаемый итератор указывает на последний переставленный элемент в целевом диапазоне.



Диапазоны не должны накладываться друг на друга

Алгоритмы обмена

```
// swap.cpp
...
#include <algorithm>

...

std::vector<int> myVec{0, 1, 2, 3, 4, 5, 6, 7, 9};
std::vector<int> myVec2(9);

std::swap(myVec, myVec2);

for (auto v: myVec) std::cout << v << " "; // 0 0 0 0 0 0 0 0 0
for (auto v: myVec2) std::cout << v << " "; // 0 1 2 3 4 5 6 7 9

std::string str{"abcdefghijklmnp"};
std::string str2{"-----"};
std::swap_ranges(str.begin(), str.begin()+5, str2.begin()+5);
std::cout << str << '\n'; // -----fghijklmnp
std::cout << str2 << '\n'; // -----abcde-----
```

Преобразование диапазонов

Стандартный алгоритм `std::transform` применяет унарную либо бинарную вызываемую единицу кода к диапазону и копирует измененные элементы в целевой диапазон.

Применить унарную вызываемую единицу кода `fun` к элементам входного диапазона с копированием результата в диапазон `result`:

```
OutIt transform(InpIt first1, InpIt last1, OutIt result, UnFun fun)
FwdIt2 transform(ExePol pol, FwdIt first1, FwdIt last1, FwdIt2 result, UnFun fun)
```

Применить бинарную вызываемую единицу кода fun к обоим входным диапазонам с копированием результата в диапазон result:

```
OutIt transform(InpIt1 first1, InpIt1 last1, InpIt2 first2, OutIt result,
               BiFun fun)
FwdIt3 transform(ExePol pol, FwdIt1 first1, FwdIt1 last1,
               FwdIt2 first2, FwdIt3 result, BiFun fun)
```

Разница между этими двумя версиями заключается в том, что в первой версии вызываемая единица кода применяется к каждому элементу диапазона, а во второй вызываемая единица кода применяется параллельно к парам элементов обоих диапазонов. Возвращаемый итератор указывает на одну позицию после последнего преобразованного элемента.

Алгоритмы преобразования

```
// transform.cpp
...
#include <algorithm>

...

std::string str{"abcdefghijklmnopqrstuvwxy"};

std::transform(str.begin(), str.end(), str.begin(),
               [](char c){ return std::toupper(c); });
std::cout << str;           // ABCDEFGHIJKLMNOPQRSTUVWXYZ

std::vector<std::string> vecStr{"Only", "for", "testing", "purpose", "." };
std::vector<std::string> vecStr2(5, "-");
std::vector<std::string> vecRes;
std::transform(vecStr.begin(), vecStr.end(),
               vecStr2.begin(), std::back_inserter(vecRes),
               [](std::string a, std::string b){ return std::string(b)+a+b; });
for (auto str: vecRes) std::cout << str << " ";
                        // / -Only- -for- -testing- -purpose- -.-
```

Инверсия диапазонов

Стандартные алгоритмы `std::reverse` и `std::reverse_copy` обращают вспять порядок следования элементов в своем диапазоне.

Инвертировать порядок следования элементов в диапазоне:

```
void reverse(BiIt first, BiIt last)
void reverse(ExePol pol, BiIt first, BiIt last)
```

Инвертировать порядок следования элементов в диапазоне с копированием результата в диапазон `result`:

```
OutIt reverse_copy(BiIt first, BiIt last, OutIt result)
FwdIt reverse_copy(ExePol pol, BiIt first, BiIt last, FwdIt result)
```

Оба алгоритма нуждаются в двунаправленных итераторах. Возвращаемый итератор указывает на выходной диапазон `result` до копирования элементов.

Алгоритмы инверсии диапазонов

```
// algorithmen.cpp
...
#include <algorithm>

...

std::string str{"123456789"};
std::reverse(str.begin(), str.begin()+5);
std::cout << str;           // 543216789
```

Поворот диапазонов

Стандартные алгоритмы `std::rotate` и `std::rotate_copy` поворачивают элементы.

Повернуть элементы таким образом, что середина становится новым первым элементом:

```
FwdIt rotate(FwdIt first, FwdIt middle, FwdIt last)
FwdIt rotate(ExePol pol, FwdIt first, FwdIt middle, FwdIt last)
```

Повернуть элементы таким образом, что середина становится новым первым элементом, с копированием результата в диапазон `result`:

```
OutIt rotate_copy(FwdIt first, FwdIt middle, FwdIt last, OutIt result)
FwdIt2 rotate_copy(ExePol pol, FwdIt first, FwdIt middle, FwdIt last,
                  FwdIt2 result)
```

Оба алгоритма нуждаются во впереднаправленных итераторах. На выходе возвращается концевой итератор на скопированном диапазоне.

Алгоритмы поворота

```
// rotate.cpp
...
#include <algorithm>
```

```

...

std::string str{"12345"};

for (auto i= 0; i < str.size(); ++i){
    std::string tmp{str};
    std::rotate(tmp.begin(), tmp.begin()+i , tmp.end());
    std::cout << tmp << " ";
}
// 12345 23451 34512 45123 51234

```

Сдвиг диапазонов

Стандартные алгоритмы стандарта C++20 `std::shift_left` и `std::shift_right` сдвигают элементы в диапазоне. Оба алгоритма возвращают впереднаправленный итератор.

Сдвинуть элементы диапазона к началу диапазона на *n* позиций с возвращением конца результирующего диапазона:

```

FwdIt shift_left(FwdIt first, FwdIt last, Num n)
FwdIt shift_left(ExePol pol, FwdIt first, FwdIt last, Num n)

```

Сдвинуть элементы диапазона к концу диапазона на *n* позиций с возвращением начала результирующего диапазона:

```

FwdIt shift_right(FwdIt first, FwdIt last, Num n)
FwdIt shift_right(ExePol pol, FwdIt first, FwdIt last, Num n)

```

Операции `std::shift_left` и `std::shift_right` не имеют эффекта, если $n == 0$ || $n >= last - first$. Элементы диапазона перемещаются.

Сдвиг элементов диапазона

```

// shiftRange.cpp
...
#include <algorithm>
...

std::vector<int> myVec{1, 2, 3, 4, 5, 6, 7};
for (auto v: myVec) std::cout << v << " "; // 1 2 3 4 5 6 7

auto newEnd = std::shift_left(myVec.begin(), myVec.end(), 2);
myVec.erase(newEnd, myVec.end());
for (auto v: myVec) std::cout << v << " "; // 3 4 5 6 7

auto newBegin = std::shift_right(myVec.begin(), myVec.end(), 2);
myVec.erase(myVec.begin(), newBegin);

```

```
for (auto v: myVec) std::cout << v << " "; // 3 4 5
```

Произвольная перетасовка диапазонов

С помощью стандартных алгоритмов `std::random_shuffle` и `std::shuffle` диапазоны можно произвольно перетасовывать.

Произвольно перетасовать элементы диапазона:

```
void random_shuffle(RanIt first, RanIt last)
```

Произвольно перетасовать элементы диапазона с использованием генератора случайных чисел `gen`:

```
void random_shuffle(RanIt first, RanIt last, RanNumGen&& gen)
```

Произвольно перетасовать элементы диапазона с использованием генератора равномерных случайных чисел `gen`:

```
void shuffle(RanIt first, RanIt last, URNG&& gen)
```

Алгоритмам нужны итераторы с произвольным доступом. Генератор `RanNumGen&& gen` должен быть вызываемой единицей кода, принимающей аргумент и возвращающей значение в пределах аргументов. Генератор `URNG&& gen` должен быть генератором равномерных случайных чисел.



Рекомендуется отдавать предпочтение алгоритму `std::shuffle`

Настоятельно рекомендуется использовать алгоритм `std::shuffle` вместо алгоритма `std::random_shuffle`. Алгоритм `std::random_shuffle` был упразднен начиная со стандарта C++14 и удален в стандарте C++17, потому что на внутреннем уровне в нем используется функция C `rand`.

Алгоритмы произвольной перетасовки

```
// shuffle.cpp
...
#include <algorithm>

...

using std::chrono::system_clock;
using std::default_random_engine;

std::vector<int> vec1{0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
std::vector<int> vec2(vec1);
```

```
std::random_shuffle(vec1.begin(), vec1.end());
for (auto v: vec1) std::cout << v << " "; // 4 3 7 8 0 5 2 1 6 9

unsigned seed= system_clock::now().time_since_epoch().count();
std::shuffle(vec2.begin(), vec2.end(), default_random_engine(seed));
for (auto v: vec2) std::cout << v << " "; // 4 0 2 3 9 6 5 1 8 7
```

Аргумент `seed` инициализирует генератор случайных чисел.

Удаление дубликатов

Благодаря стандартным алгоритмам `std::unique` и `std::unique_copy` в C++ появилось больше возможностей для удаления смежных дубликатов. Это можно делать как с бинарным предикатом, так и без него.

Удалить смежные дубликаты:

```
FwdIt unique(FwdIt first, FwdIt last)
FwdIt unique(ExePol pol, FwdIt first, FwdIt last)
```

Удалить смежные дубликаты, удовлетворяющие бинарному предикату:

```
FwdIt unique(FwdIt first, FwdIt last, BiPred pre)
FwdIt unique(ExePol pol, FwdIt first, FwdIt last, BiPred pre)
```

Удалить смежные дубликаты с копированием результата в диапазон `result`:

```
OutIt unique_copy(InpIt first, InpIt last, OutIt result)
FwdIt2 unique_copy(ExePol pol, FwdIt first, FwdIt last, FwdIt2 result)
```

Удалить смежные дубликаты, удовлетворяющие бинарному предикату, с копированием результата в диапазон `result`:

```
OutIt unique_copy(InpIt first, InpIt last, OutIt result, BiPred pre)
FwdIt2 unique_copy(ExePol pol, FwdIt first, FwdIt last,
                  FwdIt2 result, BiPred pre)
```



Алгоритмы с префиксом `unique` возвращают новый логический концевой итератор

Алгоритмы с префиксом `unique` возвращают логический концевой итератор диапазона. Элементы должны удаляться, используя идиому «стереть-удалить».

Алгоритмы удаления дубликатов

```
// removeDuplicates.cpp
...
#include <algorithm>
```

...

```

std::vector<int> myVec{0, 0, 1, 1, 2, 2, 3, 4, 4, 5,
                      3, 6, 7, 8, 1, 3, 3, 8, 8, 9};

myVec.erase(std::unique(myVec.begin(), myVec.end()), myVec.end());
for (auto v: myVec) std::cout << v << " "; // 0 1 2 3 4 5 3 6 7 8 1 3 8 9

std::vector<int> myVec2{1, 4, 3, 3, 3, 5, 7, 9, 2, 4, 1, 6, 8,
                       0, 3, 5, 7, 8, 7, 3, 9, 2, 4, 2, 5, 7, 3};
std::vector<int> resVec;
resVec.reserve(myVec2.size());
std::unique_copy(myVec2.begin(), myVec2.end(), std::back_inserter(resVec),
                [](int a, int b){ return (a%2) == (b%2); } );
for(auto v: myVec2) std::cout << v << " ";
// 1 4 3 3 3 5 7 9 2 4 1 6 8 0 3 5 7 8 7 3 9 2 4 2 5 7 3
for(auto v: resVec) std::cout << v << " "; // 1 4 3 2 1 6 3 8 7 2 5

```

Разбиение диапазонов



Что такое разбиение в контексте множества?

Разбиение множества – это разложение множества на подмножества таким образом, что каждый элемент множества входит строго в одно подмножество. Бинарный предикат определяет подмножество таким образом, что члены первого подмножества удовлетворяют предикату. Остальные элементы находятся во втором подмножестве.

C++ предлагает несколько стандартных алгоритмов для работы с разбиением диапазонов. Все они нуждаются в унарном предикате `pre`. Алгоритмы `std::partition` и `std::stable_partition` разбивают диапазон и возвращают точку разбиения. С помощью алгоритма `std::partition_point` можно получать точку разбиения. После этого с помощью алгоритма `std::is_partitioned` можно проверять раздел диапазона, а с помощью алгоритма `std::partition_copy` – его копировать.

Выполнить проверку, был ли диапазон разбит на разделы:

```

bool is_partitioned(InpIt first, InpIt last, UnPre pre)
bool is_partitioned(ExePol pol, FwdIt first, FwdIt last, UnPre pre)

```

Разбить диапазон на разделы:

```

FwdIt partition(FwdIt first, FwdIt last, UnPre pre)
FwdIt partition(ExePol pol, FwdIt first, FwdIt last, UnPre pre)

```

Разбить диапазон (стабильное разбиение):

```
BiIt stable_partition(FwdIt first, FwdIt last, UnPre pre)
BiIt stable_partition(ExePol pol, FwdIt first, FwdIt last, UnPre pre)
```

Скопировать раздел в двух диапазонах:

```
pair<OutIt1, OutIt2> partition_copy(InIt first, InIt last,
                                   OutIt1 result_true, OutIt2 result_false, UnPre pre)
pair<FwdIt1, FwdIt2> partition_copy(ExePol pol, FwdIt1 first, FwdIt1 last,
                                   FwdIt2 result_true, FwdIt3 result_false, UnPre pre)
```

Возвратить точку разбиения:

```
FwdIt partition_point(FwdIt first, FwdIt last, UnPre pre)
```

В отличие от алгоритма `std::partition`, алгоритм `std::stable_partition` гарантирует, что элементы сохраняют свой относительный порядок. Возвращаемые итераторы `FwdIt` и `BiIt` указывают на второе подмножество начальной позиции раздела. Объект `std::pair<OutIt, OutIt>` алгоритма `std::partition_copy` содержит концевой итератор подмножеств `result_true` и `result_false`. Поведение алгоритма `std::partition_point` не определено, если диапазон не разбит на разделы.

Алгоритмы разбиения

```
// partition.cpp
...
#include <algorithm>

...

using namespace std;

bool isOdd(int i){ return (i%2) == 1; }
vector<int> vec{1, 4, 3, 4, 5, 6, 7, 3, 4, 5, 6, 0, 4,
               8, 4, 6, 6, 5, 8, 8, 3, 9, 3, 7, 6, 4, 8};

auto parPoint= partition(vec.begin(), vec.end(), isOdd);
for (auto v: vec) cout << v << " ";
// 1 7 3 3 5 9 7 3 3 5 5 0 4 8 4 6 6 6 8 8 4 6 4 4 6 4 8
for (auto v= vec.begin(); v != parPoint; ++v) cout << *v << " ";
// 1 7 3 3 5 9 7 3 3 5 5
for (auto v= parPoint; v != vec.end(); ++v) cout << *v << " ";
// 4 8 4 6 6 6 8 8 4 6 4 4 6 4 8

cout << is_partitioned(vec.begin(), vec.end(), isOdd); // true

list<int> le;
list<int> ri;
```

```
partition_copy(vec.begin(), vec.end(), back_inserter(le), back_inserter(ri),
              [](int i) { return i < 5; });
for (auto v: le) cout << v << " "; // 1 3 3 3 3 0 4 4 4 4 4 4
for (auto v: ri) cout << v << " "; // 7 5 9 7 5 5 8 6 6 6 8 8 6 6 8
```

Сортировка

С помощью стандартных алгоритмов `std::sort` и `std::stable_sort` диапазон можно сортировать, а с помощью стандартного алгоритма `std::partial_sort` – сортировать его вплоть до определенной позиции. Дополнительно к этому стандартный алгоритм `std::partial_sort_copy` копирует частично отсортированный диапазон. С помощью стандартного алгоритма `std::nth_element` можно назначать элементу отсортированную позицию в диапазоне. С помощью стандартного алгоритма `std::is_sorted` можно выполнять проверку на отсортированность диапазона. Если требуется узнать, до какой позиции диапазон отсортирован, то рекомендуется использовать стандартный алгоритм `std::is_sorted_until`.

По умолчанию в качестве критерия сортировки используется предопределенный функциональный объект `std::less`. Однако можно использовать свой собственный критерий сортировки. Он должен подчиняться строго слабому упорядочиванию.

Отсортировать элементы диапазона:

```
void sort(RaIt first, RaIt last)
void sort(ExePol pol, RaIt first, RaIt last)

void sort(RaIt first, RaIt last, BiPre pre)
void sort(ExePol pol, RaIt first, RaIt last, BiPre pre)
```

Отсортировать элементы диапазона (стабильная сортировка):

```
void stable_sort(RaIt first, RaIt last)
void stable_sort(ExePol pol, RaIt first, RaIt last)

void stable_sort(RaIt first, RaIt last, BiPre pre)
void stable_sort(ExePol pol, RaIt first, RaIt last, BiPre pre)
```

Частично отсортировать элементы диапазона вплоть до середины:

```
void partial_sort(RaIt first, RaIt middle, RaIt last)
void partial_sort(ExePol pol, RaIt first, RaIt middle, RaIt last)

void partial_sort(RaIt first, RaIt middle, RaIt last, BiPre pre)
void partial_sort(ExePol pol, RaIt first, RaIt middle, RaIt last, BiPre pre)
```

Частично отсортировать элементы диапазона с их копированием в целевые диапазоны `result_first` и `result_last`:

```

RaIt partial_sort_copy(InIt first, InIt last,
                      RaIt result_first, RaIt result_last)
RaIt partial_sort_copy(ExePol pol, FwdIt first, FwdIt last,
                      RaIt result_first, RaIt result_last)

RaIt partial_sort_copy(InIt first, InIt last,
                      RaIt result_first, RaIt result_last, BiPre pre)
RaIt partial_sort_copy(ExePol pol, FwdIt first, FwdIt last,
                      RaIt result_first, RaIt result_last, BiPre pre)

```

Выполнить проверку диапазона на отсортированность:

```

bool is_sorted(FwdIt first, FwdIt last)
bool is_sorted(ExePol pol, FwdIt first, FwdIt last)

bool is_sorted(FwdIt first, FwdIt last, BiPre pre)
bool is_sorted(ExePol pol, FwdIt first, FwdIt last, BiPre pre)

```

Возвратить позицию первого элемента, который не удовлетворяет критерию сортировки:

```

FwdIt is_sorted_until(FwdIt first, FwdIt last)
FwdIt is_sorted_until(ExePol pol, FwdIt first, FwdIt last)

FwdIt is_sorted_until(FwdIt first, FwdIt last, BiPre pre)
FwdIt is_sorted_until(ExePol pol, FwdIt first, FwdIt last, BiPre pre)

```

Упорядочить диапазон таким образом, чтобы n-й элемент занимал правильную (отсортированную) позицию:

```

void nth_element(RaIt first, RaIt nth, RaIt last)
void nth_element(ExePol pol, RaIt first, RaIt nth, RaIt last)

void nth_element(RaIt first, RaIt nth, RaIt last, BiPre pre)
void nth_element(ExePol pol, RaIt first, RaIt nth, RaIt last, BiPre pre)

```

Ниже приведен фрагмент исходного кода.

Алгоритмы сортировки

```

// sort.cpp
...
#include <algorithm>
...

std::string str{"RUdAjDkaACsdfjwldXmnEiVSEZTiepf0Ikue"};

std::cout << std::is_sorted(str.begin(), str.end()); // false

```

```

std::partial_sort(str.begin(), str.begin()+30, str.end());
std::cout << str;           // AACDEEIORSTUVXZaddddeeffgiijkwspnmluk

auto sortUntil= std::is_sorted_until(str.begin(), str.end());
std::cout << *sortUntil;    // s
for (auto charIt= str.begin(); charIt != sortUntil; ++charIt)
    std::cout << *charIt;   // AACDEEIORSTUVXZaddddeeffgiijkw

std::vector<int> vec{1, 0, 4, 3, 5};
auto vecIt= vec.begin();

while(vecIt != vec.end()){
    std::nth_element(vec.begin(), vecIt++, vec.end());
    std::cout << std::distance(vec.begin(), vecIt) << "-th ";
    for (auto v: vec) std::cout << v << "/";
}
// 1-th 01435/2-th 01435/3-th 10345/4-th 30145/5-th 10345

```

Двоичный поиск

В стандартных алгоритмах двоичного поиска используются уже отсортированные диапазоны. Для поиска элемента следует использовать стандартный алгоритм `std::binary_search`. При использовании стандартного алгоритма `std::lower_bound` будет получен итератор для первого элемента, который не меньше заданного значения. При использовании стандартного алгоритма `std::upper_bound` будет получен итератор для первого элемента, который больше заданного значения. Стандартный алгоритм `std::equal_range` сочетает в себе оба алгоритма.

Если контейнер состоит из n элементов, то для поиска требуется в среднем $\log_2(n)$ сравнений. Двоичный поиск нуждается в использовании того же критерия сравнения, который применялся для сортировки контейнера. По умолчанию используется критерий сравнения `std::less`, но его можно отрегулировать. Пользовательский критерий сортировки должен подчиняться строго слабому упорядочиванию. В противном случае результат программ будет неопределенным.

Функции-члены неупорядоченного ассоциативного контейнера в общем случае работают быстрее.

Отыскать значение `val` в диапазоне:

```

bool binary_search(FwdIt first, FwdIt last, const T& val)
bool binary_search(FwdIt first, FwdIt last, const T& val, BiPre pre)

```

Возвратить позицию первого элемента диапазона, который не меньше значения `val`:

```

FwdIt lower_bound(FwdIt first, FwdIt last, const T& val)
FwdIt lower_bound(FwdIt first, FwdIt last, const T& val, BiPre pre)

```

Возвратить позицию первого элемента диапазона, который больше значения `val`:

```
FwdIt upper_bound(FwdIt first, FwdIt last, const T& val)
FwdIt upper_bound(FwdIt first, FwdIt last, const T& val, BiPre pre)
Возвратить пару std::lower_bound и std::upper_bound для значения val:
pair<FwdIt, FwdIt> equal_range(FwdIt first, FwdIt last, const T& val)
pair<FwdIt, FwdIt> equal_range(FwdIt first, FwdIt last, const T& val, BiPre pre)
```

Наконец, ниже приведен фрагмент исходного кода.

Алгоритмы двоичного поиска

```
// binarySearch.cpp
...
#include <algorithm>

...

using namespace std;

bool isLessAbs(int a, int b){
    return abs(a) < abs(b);
}

vector<int> vec{-3, 0, -3, 2, -3, 5, -3, 7, -0, 6, -3, 5,
               -6, 8, 9, 0, 8, 7, -7, 8, 9, -6, 3, -3, 2};

sort(vec.begin(), vec.end(), isLessAbs);
for (auto v: vec) cout << v << " ";
    // 0 0 0 2 2 -3 -3 -3 -3 -3 3 -3 5 5 -6 -6 6 7 -7 7 8 8 8 9 9
cout << binary_search(vec.begin(), vec.end(), -5, isLessAbs); // true
cout << binary_search(vec.begin(), vec.end(), 5, isLessAbs);   // true

auto pair= equal_range(vec.begin(), vec.end(), 3, isLessAbs);
cout << distance(vec.begin(), pair.first);    // 5
cout << distance(vec.begin(), pair.second)-1; // 11

for (auto threeIt= pair.first; threeIt != pair.second; ++threeIt)
    cout << *threeIt << " ";                // -3 -3 -3 -3 -3 3 -3
```

Операции слияния

Операции слияния позволяют выполнять слияние отсортированных диапазонов в новый отсортированный диапазон. Стандартный алгоритм нуждается в том, чтобы диапазоны и алгоритм использовали одинаковый критерий сор-

тировки. В противном случае программа будет вести себя неопределенно. По умолчанию используется предопределенный критерий сортировки `std::less`. При использовании другого критерия сортировки он должен подчиняться строго слабому упорядочиванию. В противном случае результат программы будет неопределенным.

С помощью стандартных алгоритмов `std::inplace_merge` и `std::merge` можно выполнять слияние двух отсортированных диапазонов. С помощью стандартного алгоритма `std::includes` можно выполнять проверку на вхождение одного отсортированного диапазона в другой отсортированный диапазон. С помощью стандартных алгоритмов `std::set_difference`, `std::set_intersection`, `std::set_symmetric_difference` и `std::set_union` можно комбинировать два отсортированных диапазона в новый отсортированный диапазон.

Выполнить слияние двух отсортированных поддиапазонов `[first, mid)` и `[mid, last)`:

```
void inplace_merge(BiIt first, BiIt mid, BiIt last)
void inplace_merge(ExePol pol, BiIt first, BiIt mid, BiIt last)

void inplace_merge(BiIt first, BiIt mid, BiIt last, BiPre pre)
void inplace_merge(ExePol pol, BiIt first, BiIt mid, BiIt last, BiPre pre)
```

Выполнить слияние двух отсортированных поддиапазонов с копированием результата в диапазон `result`:

```
OutIt merge(InpIt first1, InpIt last1, InpIt first2, InpIt last2, OutIt result)
FwdIt3 merge(ExePol pol, FwdIt1 first1, FwdIt1 last1,
             FwdIt2 first2, FwdIt2 last2, FwdIt3 result)

OutIt merge(InpIt first1, InpIt last1, InpIt first2, InpIt last2, OutIt result,
            BiPre pre)
FwdIt3 merge(ExePol pol, FwdIt1 first1, FwdIt1 last1,
            FwdIt2 first2, FwdIt2 last2, FwdIt3 result, BiPre pre)
```

Выполнить проверку, что все элементы второго диапазона находятся в первом диапазоне:

```
bool includes(InpIt first1, InpIt last1, InpIt1 first2, InpIt1 last2)
bool includes(ExePol pol, FwdIt first1, FwdIt last1, FwdIt1 first2, FwdIt1 last2)

bool includes(InpIt first1, InpIt last1, InpIt first2, InpIt last2, BinPre pre)
bool includes(ExePol pol, FwdIt first1, FwdIt last1,
            FwdIt1 first2, FwdIt1 last2, BinPre pre)
```

Скопировать элементы первого диапазона, не находящиеся во втором диапазоне, в `result`:

```
OutIt set_difference(InpIt first1, InpIt last1, InpIt1 first2, InpIt2 last2,
                   OutIt result)
FwdIt2 set_difference(ExePol pol, FwdIt first1, FwdIt last1,
```

```
FwdIt1 first2, FwdIt1 last2, FwdIt2 result)
```

```
OutIt set_difference(InpIt first1, InpIt last1, InpIt1 first2, InpIt2 last2,  
                    OutIt result, BiPre pre)
```

```
FwdIt2 set_difference(ExePol pol, FwdIt first1, FwdIt last1,  
                    FwdIt1 first2, FwdIt1 last2, FwdIt2 result, BiPre pre)
```

Определить теоретико-множественное пересечение первого и второго диапазонов с копированием результата в диапазон result:

```
OutIt set_intersection(InpIt first1, InpIt last1, InpIt1 first2, InpIt2 last2,  
                      OutIt result)
```

```
FwdIt2 set_intersection(ExePol pol, FwdIt first1, FwdIt last1,  
                      FwdIt1 first2, FwdIt1 last2, FwdIt2 result)
```

```
OutIt set_intersection(InpIt first1, InpIt last1, InpIt1 first2, InpIt2 last2,  
                      OutIt result, BiPre pre)
```

```
FwdIt2 set_intersection(ExePol pol, FwdIt first1, FwdIt last1,  
                      FwdIt1 first2, FwdIt1 last2, FwdIt2 result, BiPre pre)
```

Определить симметрическую разность первого и второго диапазонов с копированием результата в диапазон result:

```
OutIt set_symmetric_difference(InpIt first1, InpIt last1,  
                              InpIt1 first2, InpIt2 last2, OutIt result)
```

```
FwdIt2 set_symmetric_difference(ExePol pol, FwdIt first1, FwdIt last1,  
                              FwdIt1 first2, FwdIt1 last2, FwdIt2 result)
```

```
OutIt set_symmetric_difference(InpIt first1, InpIt last1,  
                              InpIt1 first2, InpIt2 last2, OutIt result,  
                              BiPre pre)
```

```
FwdIt2 set_symmetric_difference(ExePol pol, FwdIt first1, FwdIt last1,  
                              FwdIt1 first2, FwdIt1 last2, FwdIt2 result,  
                              BiPre pre)
```

Определить теоретико-множественное объединение первого и второго диапазонов с копированием результата в диапазон result:

```
OutIt set_union(InpIt first1, InpIt last1,  
               InpIt1 first2, InpIt2 last2, OutIt result)
```

```
FwdIt2 set_union(ExePol pol, FwdIt first1, FwdIt last1,  
               FwdIt1 first2, FwdIt1 last2, FwdIt2 result)
```

```
OutIt set_union(InpIt first1, InpIt last1,  
               InpIt1 first2, InpIt2 last2, OutIt result, BiPre pre)
```

```
FwdIt2 set_union(ExePol pol, FwdIt first1, FwdIt last1,  
               FwdIt1 first2, FwdIt1 last2, FwdIt2 result, BiPre pre)
```

На выходе возвращается концевой итератор на целевом диапазоне. Целевой диапазон алгоритма `std::set_difference` содержит все элементы первого диапазона, но не второго. Напротив, целевой диапазон алгоритма `std::symmetric_difference` содержит только те элементы, которые являются элементами одного диапазона, но не обоих. Алгоритм `std::union` определяет теоретико-множественное объединение обоих отсортированных диапазонов.

Алгоритмы слияния

```
// merge.cpp
...
#include <algorithm>

...

std::vector<int> vec1{1, 1, 4, 3, 5, 8, 6, 7, 9, 2};
std::vector<int> vec2{1, 2, 3};

std::sort(vec1.begin(), vec1.end());
std::vector<int> vec(vec1);

vec1.reserve(vec1.size() + vec2.size());
vec1.insert(vec1.end(), vec2.begin(), vec2.end());
for (auto v: vec1) std::cout << v << " "; // 1 1 2 3 4 5 6 7 8 9 1 2 3

std::inplace_merge(vec1.begin(), vec1.end()-vec2.size(), vec1.end());
for (auto v: vec1) std::cout << v << " "; // 1 1 1 2 2 3 3 4 5 6 7 8 9

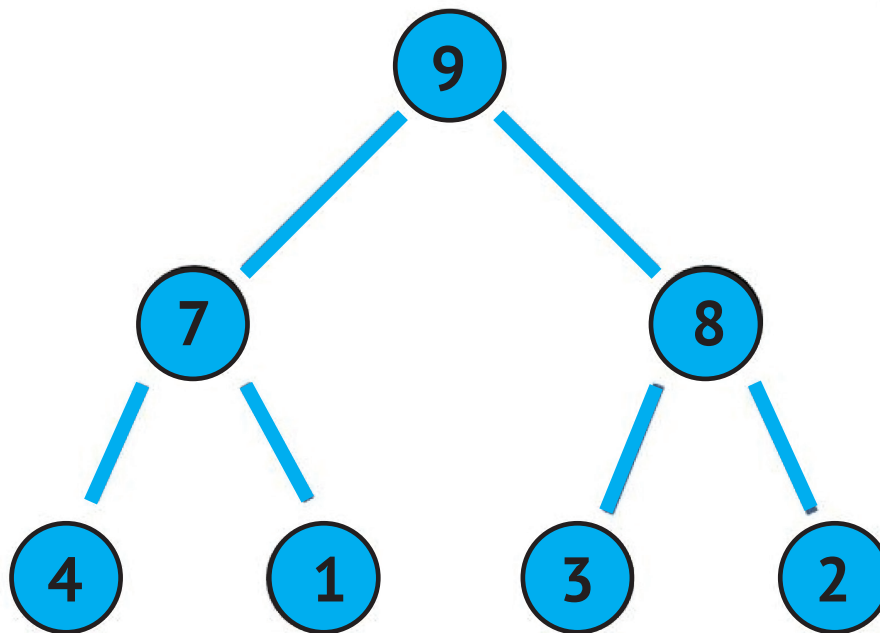
vec2.push_back(10);
for (auto v: vec) std::cout << v << " "; // 1 1 2 3 4 5 6 7 8 9
for (auto v: vec2) std::cout << v << " "; // 1 2 3 10

std::vector<int> res;
std::set_symmetric_difference(vec.begin(), vec.end(), vec2.begin(), vec2.end(),
                             std::back_inserter(res));

for (auto v : res) std::cout << v << " "; // 1 4 5 6 7 8 9 10

res= {};
std::set_union(vec.begin(), vec.end(), vec2.begin(), vec2.end(),
std::back_inserter(res));
for (auto v : res) std::cout << v << " "; // 1 1 2 3 4 5 6 7 8 9 10
```

Кучи



Что такое куча?

Куча – это дерево двоичного поиска, родительские элементы которого всегда больше дочерних элементов. Деревья-кучи оптимизированы под эффективную сортировку элементов.

С помощью стандартного алгоритма `std::make_heap` можно создавать кучу. С помощью стандартного алгоритма `std::push_heap` можно заталкивать новые элементы в кучу. Напротив, с помощью стандартного алгоритма `std::pop_heap` можно извлекать из кучи самый большой элемент. Обе операции учитывают характеристики кучи. Алгоритм `std::push_heap` перемещает последний элемент диапазона в кучу; алгоритм `std::pop_heap` перемещает самый большой элемент кучи на последнюю позицию в диапазоне. С помощью алгоритма `std::is_heap` можно проверять, является ли диапазон кучей. С помощью алгоритма `std::is_heap_until` можно определять, до какой позиции диапазон является кучей. Стандартный алгоритм `std::sort_heap` сортирует кучу.

Алгоритмы кучи нуждаются в наличии у диапазона и алгоритма одинакового критерия сортировки. В противном случае программа будет вести себя неопределенно. По умолчанию используется предопределенный критерий сортировки `std::less`.

Если используется пользовательский критерий сортировки, то он должен подчиняться строго слабому упорядочиванию. В противном случае программа будет иметь неопределенное поведение.

Создать кучу из указанного диапазона:

```
void make_heap(RaIt first, RaIt last)
void make_heap(RaIt first, RaIt last, BiPre pre)
```

Проверить, является ли диапазон кучей:

```
bool is_heap(RaIt first, RaIt last)
bool is_heap(ExePol pol, RaIt first, RaIt last)
```

```
bool is_heap(RaIt first, RaIt last, BiPre pre)
bool is_heap(ExePol pol, RaIt first, RaIt last, BiPre pre)
```

Определить позицию, до которой диапазон является кучей:

```
RaIt is_heap_until(RaIt first, RaIt last)
RaIt is_heap_until(ExePol pol, RaIt first, RaIt last)
```

```
RaIt is_heap_until(RaIt first, RaIt last, BiPre pre)
RaIt is_heap_until(ExePol pol, RaIt first, RaIt last, BiPre pre)
```

Отсортировать кучу:

```
void sort_heap(RaIt first, RaIt last)
void sort_heap(RaIt first, RaIt last, BiPre pre)
```

Переместить последний элемент диапазона в кучу. Диапазон `[first, last-1)` должен быть кучей:

```
void push_heap(RaIt first, RaIt last)
void push_heap(RaIt first, RaIt last, BiPre pre)
```

Удалить самый большой элемент из кучи и поместить его в конец диапазона:

```
void pop_heap(RaIt first, RaIt last)
void pop_heap(RaIt first, RaIt last, BiPre pre)
```

С помощью алгоритма `std::pop_heap` можно удалять из кучи самый большой элемент. После этого самый большой элемент становится последним элементом диапазона. Для удаления элемента из кучи `h` используется функция-член `h.pop_back`.

Алгоритмы кучи

```
// heap.cpp
...
#include <algorithm>
...

std::vector<int> vec{4, 3, 2, 1, 5, 6, 7, 9, 10};

std::make_heap(vec.begin(), vec.end());

for (auto v: vec) std::cout << v << " "; // 10 9 7 4 5 6 2 3 1
```

```
std::cout << std::is_heap(vec.begin(), vec.end());           // true

vec.push_back(100);
std::cout << std::is_heap(vec.begin(), vec.end());           // false
std::cout << *std::is_heap_until(vec.begin(), vec.end());     // 100
for (auto v: vec) std::cout << v << " "; // 10 9 7 4 5 6 2 3 1 100

std::push_heap(vec.begin(), vec.end());
std::cout << std::is_heap(vec.begin(), vec.end());           // true
for (auto v: vec) std::cout << v << " "; // 100 10 7 4 9 6 2 3 1 5

std::pop_heap(vec.begin(), vec.end());
for (auto v: vec) std::cout << v << " "; // 10 9 7 4 5 6 2 3 1 100
std::cout << *std::is_heap_until(vec.begin(), vec.end());     // 100

vec.resize(vec.size()-1);
std::cout << std::is_heap(vec.begin(), vec.end());           // true
std::cout << vec.front() << '\n'; // 10
```

Минимум и максимум

С помощью стандартных алгоритмов `std::min_element`, `std::max_element` и `std::minmax_element` можно определять минимальный элемент, максимальный элемент, пару с минимальным и максимальным элементами диапазона. Каждый алгоритм может вызываться с использованием бинарного предиката. Вдобавок стандарт C++17 позволяет зажимать значение между парой граничных значений.

Возвратить минимальный элемент диапазона:

```
constexpr FwdIt min_element(FwdIt first, FwdIt last)
FwdIt min_element(ExePol pol, FwdIt first, FwdIt last)
```

```
constexpr FwdIt min_element(FwdIt first, FwdIt last, BinPre pre)
FwdIt min_element(ExePol pol, FwdIt first, FwdIt last, BinPre pre)
```

Возвратить максимальный элемент диапазона:

```
constexpr FwdIt max_element(FwdIt first, FwdIt last)
FwdIt max_element(ExePol pol, FwdIt first, FwdIt last)
```

```
constexpr FwdIt max_element(FwdIt first, FwdIt last, BinPre pre)
FwdIt max_element(ExePol pol, FwdIt first, FwdIt last, BinPre pre)
```

Возвратить пару `std::min_element` и `std::max_element` диапазона:

```
constexpr pair<FwdIt, FwdIt> minmax_element(FwdIt first, FwdIt last)
```

```
pair<FwdIt, FwdIt> minmax_element(ExePol pol, FwdIt first, FwdIt last)
```

```
constexpr pair<FwdIt, FwdIt> minmax_element(FwdIt first, FwdIt last, BinPre pre)
pair<FwdIt, FwdIt> minmax_element(ExePol pol, FwdIt first, FwdIt last,
                                BinPre pre)
```

Если диапазон имеет более одного минимального или максимального элемента, то возвращается первый.

Алгоритмы вычисления минимума и максимума

```
// minMax.cpp
...
#include <algorithm>

...

int toInt(const std::string& s){
    std::stringstream buff;
    buff.str("");
    buff << s;
    int value;
    buff >> value;
    return value;
}

std::vector<std::string> myStrings{"94", "5", "39", "-4", "-49", "1001", "-77",
                                   "23", "0", "84", "59", "96", "6", "-94"};
auto str= std::minmax_element(myStrings.begin(), myStrings.end());

std::cout << *str.first << ":" << *str.second;    // -4:96

auto asInt= std::minmax_element(myStrings.begin(), myStrings.end(),
                                [](std::string a, std::string b){ return toInt(a) < toInt(b); });
std::cout << *asInt.first << ":" << *asInt.second; // -94:1001
```

Стандартный алгоритм `std::clamp` зажимает значение между парой граничных значений:

```
constexpr const T& clamp(const T& v, const T& lo, const T& hi);
constexpr const T& clamp(const T& v, const T& lo, const T& hi, BinPre pre);
```

Он возвращает ссылку на элемент `v`, если элемент `lo` меньше элемента `v` и элемент `v` меньше элемента `hi`. В противном случае возвращается ссылка на элемент `lo` либо на элемент `hi`. По умолчанию в качестве критерия сравнения используется оператор `<`, но также можно указывать бинарный предикат `pre`.

Зажатие значения между двумя границами

```
// clamp.cpp
...
#include <algorithm>

...

auto values = {1, 2, 3, 4, 5, 6, 7};
for (auto v: values) std::cout << v << ' '; // 1 2 3 4 5 6 7

auto lo = 3;
auto hi = 6;
for (auto v: values) std::cout << std::clamp(v, lo, hi) << ' '; // 3 3 3 4 5 6 6
```

Перестановки

Стандартные алгоритмы `std::prev_permutation` и `std::next_permutation` возвращают предыдущую меньшую или следующую бóльшую перестановку нового упорядоченного диапазона. Если меньшая или бóльшая перестановка недоступна, то алгоритмы возвращают `false`. Оба алгоритма нуждаются в двунаправленных итераторах. По умолчанию используется предопределенный критерий сортировки `std::less`. Если используется пользовательский критерий сортировки, то он должен подчиняться строго слабому упорядочиванию. В противном случае программа будет вести себя неопределенно.

Применить предыдущую перестановку к диапазону:

```
bool prev_permutation(BiIt first, BiIt last)
bool prev_permutation(BiIt first, BiIt last, BiPred pre)
```

Применить следующую перестановку к диапазону:

```
bool next_permutation(BiIt first, BiIt last)
bool next_permutation(BiIt first, BiIt last, BiPred pre)
```

С помощью обоих алгоритмов можно генерировать все перестановки диапазона.

Алгоритмы перестановки

```
// permutation.cpp
...
#include <algorithm>

...

std::vector<int> myInts{1, 2, 3};
```

```
do{
    for (auto i: myInts) std::cout << i;
    std::cout << " ";
} while(std::next_permutation(myInts.begin(), myInts.end()));
// 123 132 213 231 312 321

std::reverse(myInts.begin(), myInts.end());
do{
    for (auto i: myInts) std::cout << i;
    std::cout << " ";
} while(std::prev_permutation(myInts.begin(), myInts.end()));
// 321 312 231 213 132 123
```

Численные алгоритмы

Численные стандартные алгоритмы `std::accumulate`, `std::adjacent_difference`, `std::partial_sum`, `std::inner_product` и `std::iota` и шесть дополнительных алгоритмов стандарта C++17 `std::exclusive_scan`, `std::inclusive_scan`, `std::transform_exclusive_scan`, `std::transform_inclusive_scan`, `std::reduce` и `std::transform_reduce` являются особыми. Все они определены в заголовке `<numeric>` и имеют широкое применение, так как могут вызываться с вызываемой единицей кода.

Накопить элементы диапазона, где `init` – это начальное значение:

```
T accumulate(InpIt first, InpIt last, T init)
T accumulate(InpIt first, InpIt last, T init, BiFun fun)
```

Вычислить разность между смежными элементами диапазона с сохранением результата в диапазоне `result`:

```
OutIt adjacent_difference(InpIt first, InpIt last, OutIt result)
FwdIt2 adjacent_difference(ExePol pol, FwdIt first, FwdIt last, FwdIt2 result)
```

```
OutIt adjacent_difference(InpIt first, InpIt last, OutIt result, BiFun fun)
FwdIt2 adjacent_difference(ExePol pol, FwdIt first, FwdIt last,
                          FwdIt2 result, BiFun fun)
```

Вычислить частичную сумму диапазона:

```
OutIt partial_sum(InpIt first, InpIt last, OutIt result)
OutIt partial_sum(InpIt first, InpIt last, OutIt result, BiFun fun)
```

Вычислить внутреннее произведение двух диапазонов и вернуть результат:

```
T inner_product(InpIt first1, InpIt last1, OutIt first2, T init)
T inner_product(InpIt first1, InpIt last1, OutIt first2, T init,
                BiFun fun1, BiFun fun2)
```

Присвоить каждому элементу диапазона `a` по одному последовательно возрастающему значению, где `val` – это начальное значение:

```
void iota(FwdIt first, FwdIt last, T val)
```

Более сложные для понимания алгоритмы

Алгоритм `std::accumulate` без вызываемой единицы кода использует следующую ниже стратегию:

```
result = init;
result += *(first+0);
result += *(first+1);
...
```

Алгоритм `std::adjacent_difference` без вызываемой единицы кода использует следующую ниже стратегию:

```
*(result) = *first;
*(result+1) = *(first+1) - *(first);
*(result+2) = *(first+2) - *(first+1);
...
```

Алгоритм `std::partial_sum` без вызываемой единицы кода использует следующую ниже стратегию:

```
*(result) = *first;
*(result+1) = *first + *(first+1);
*(result+2) = *first + *(first+1) + *(first+2);
...
```

Вариация сложного алгоритма `inner_product(InpIt, InpIt, OutIt, T, BiFun fun1, BiFun fun2)` с двумя бинарными вызываемыми единицами кода использует следующую стратегию: вторая вызываемая единица кода `fun2` применяется к каждой паре диапазонов, чтобы создать временный целевой диапазон `tmp`, а первая вызываемая единица кода применяется к каждому элементу целевого диапазона `tmp`, чтобы их накапливать и, соответственно, генерировать окончательный результат.

Численные алгоритмы

```
// numeric.cpp
...
#include <numeric>

...

std::array<int, 9> arr{1, 2, 3, 4, 5, 6, 7, 8, 9};
std::cout << std::accumulate(arr.begin(), arr.end(), 0);           // 45
std::cout << std::accumulate(arr.begin(), arr.end(), 1,
```

```

        [](int a, int b){ return a*b; } ); // 362880

std::vector<int> vec{1, 2, 3, 4, 5, 6, 7, 8, 9};
std::vector<int> myVec;
std::adjacent_difference(vec.begin(), vec.end(),
    std::back_inserter(myVec), [](int a, int b){ return a*b; });
for (auto v: myVec) std::cout << v << " "; // 1 2 6 12 20 30 42 56 72
std::cout << std::inner_product(vec.begin(), vec.end(), arr.begin(), 0); // 285

myVec = {};
std::partial_sum(vec.begin(), vec.end(), std::back_inserter(myVec));
for (auto v: myVec) std::cout << v << " "; // 1 3 6 10 15 21 28 36 45

std::vector<int> myLongVec(10);
std::iota(myLongVec.begin(), myLongVec.end(), 2000);
for (auto v: myLongVec) std::cout << v << " ";
// 2000 2001 2002 2003 2004 2005 2006 2007 2008 2009

```

Новые параллельные алгоритмы в стандарте C++17

Шесть новых алгоритмов, широко используемых для параллельного исполнения, называются префиксной суммой¹. Поведение указанных алгоритмов не определено, если заданные бинарные вызываемые единицы кода не являются ассоциативными и коммутативными.

Алгоритм `reduce` редуцирует элементы диапазона, где `init` – это начальное значение. Он ведет себя так же, как алгоритм `std::accumulate`, но диапазон может перестраиваться.

```

ValType reduce(InpIt first, InpIt last)
ValType reduce(ExePol pol, InpIt first, InpIt last)

T reduce(InpIt first, InpIt last, T init)
T reduce(ExePol pol, InpIt first, InpIt last, T init)

T reduce(InpIt first, InpIt last, T init, BiFun fun)
T reduce(ExePol pol, InpIt first, InpIt last, T init, BiFun fun)

```

Алгоритм `transform_reduce` преобразовывает один или два диапазона и редуцирует их элементы (выполняет приведение), где `init` – это начальное значение. Его поведение аналогично алгоритму `std::inner_product`, но диапазон может перестраиваться.

Если при применении к двум диапазонам вызываемые единицы кода не указаны, то для преобразования диапазонов в один диапазон используется умножение, а для редукции промежуточного диапазона в результат используется сложение.

¹ См. https://en.wikipedia.org/wiki/Prefix_sum.

Если же вызываемые единицы кода указаны, то вызываемая единица кода `fun1` используется на шаге преобразования, а вызываемая единица кода `fun2` – на шаге редукции. При применении к одному диапазону вызываемая единица кода `fun2` используется для преобразования заданного диапазона.

```
T transform_reduce(InpIt first, InpIt last, InpIt first2, T init)
T transform_reduce(InpIt first, InpIt last,
                  InpIt first2, T init, BiFun fun1, BiFun fun2)

T transform_reduce(FwdIt first, FwdIt last, FwdIt first2, T init)
T transform_reduce(ExePol pol, FwdIt first, FwdIt last,
                  FwdIt first2, T init, BiFun fun1, BiFun fun2)

T transform_reduce(InpIt first, InpIt last, T init, BiFun fun1, UnFun fun2)
T transform_reduce(ExePol pol, FwdIt first, FwdIt last,
                  T init, BiFun fun1, UnFun fun2)
```



MapReduce в стандарте C++17

Функция языка Haskell¹ `map` называется в C++ `std::transform`. Если в имени алгоритма `std::transform_reduce` вместо `transform` подставить `map`, то получится `std::map_reduce`. MapReduce² – это широко известный параллельный фреймворк, который сначала соотносит каждое значение с новым значением, а затем во второй фазе редуцирует (сводит) все значения в результат.

Данный алгоритм применим напрямую в стандарте C++17. В показанном ниже примере в фазе соотнесения (`map`) каждое слово соотносится с его длиной, а в фазе редукции (`reduce`) длина всех слов редуцируется в их сумму. Результатом является сумма длин всех слов.

```
std::vector<std::string> str{"Only", "for", "testing", "purpose"};

std::size_t result= std::transform_reduce(std::execution::par,
                                          str.begin(), str.end(), 0,
                                          [](std::size_t a, std::size_t b){ return a + b; },
                                          [](std::string s){ return s.length(); });

std::cout << result << '\n';      // 21
```

Алгоритм `exclusive_scan` вычисляет исключаящую префиксную сумму с использованием бинарной операции. Он ведет себя аналогично алгоритму `std::reduce`, но предоставляет диапазон всех префиксных сумм и на каждой итерации исключает последний элемент.

```
OutIt exclusive_scan(InpIt first, InpIt last, OutIt first, T init)
FwdIt2 exclusive_scan(ExePol pol, FwdIt first, FwdIt last,
```

¹ См. <https://www.haskell.org/>.

² См. <https://en.wikipedia.org/wiki/MapReduce>.

```
FwdIt2 first2, T init)
```

```
OutIt exclusive_scan(InpIt first, InpIt last, OutIt first, T init, BiFun fun)
FwdIt2 exclusive_scan(ExePol pol, FwdIt first, FwdIt last,
    FwdIt2 first2, T init, BiFun fun)
```

Алгоритм `inclusive_scan` вычисляет включающую префиксную сумму с использованием бинарной операции. Он ведет себя аналогично алгоритму `std::reduce`, но предоставляет диапазон всех префиксных сумм и на каждой итерации включает последний элемент.

```
OutIt inclusive_scan(InpIt first, InpIt last, OutIt first2)
FwdIt2 inclusive_scan(ExePol pol, FwdIt first, FwdIt last, FwdIt2 first2)
```

```
OutIt inclusive_scan(InpIt first, InpIt last, OutIt first, BiFun fun)
FwdIt2 inclusive_scan(ExePol pol, FwdIt first, FwdIt last,
    FwdIt2 first2, BiFun fun)
```

```
OutIt inclusive_scan(InpIt first, InpIt last, OutIt first2, BiFun fun, T init)
FwdIt2 inclusive_scan(ExePol pol, FwdIt first, FwdIt last,
    FwdIt2 first2, BiFun fun, T init)
```

Алгоритм `transform_exclusive_scan` сначала преобразовывает каждый элемент, а затем вычисляет исключающие префиксные суммы.

```
OutIt transform_exclusive_scan(InpIt first, InpIt last, OutIt first2,
    T init,
    BiFun fun, UnFun fun2)
```

```
FwdIt2 transform_exclusive_scan(ExePol pol, FwdIt first, FwdIt last,
    FwdIt2 first2,
    T init,
    BiFun fun, UnFun fun2)
```

Алгоритм `transform_inclusive_scan` сначала преобразовывает каждый элемент входного диапазона, а затем вычисляет включающие префиксные суммы.

```
OutIt transform_inclusive_scan(InpIt first, InpIt last, OutIt first2,
    BiFun fun, UnFun fun2)
```

```
FwdIt2 transform_inclusive_scan(ExePol pol, FwdIt first, FwdIt last,
    FwdIt2 first2,
    BiFun fun, UnFun fun2)
```

```
OutIt transform_inclusive_scan(InpIt first, InpIt last, OutIt first2,
    BiFun fun, UnFun fun2,
    T init)
```

FwdIt first2,

с использованием политики параллельного исполнения.

Новые алгоритмы

```
// newAlgorithms.cpp
```

• • •

```
#include <execution>
```

```
#include <numeric>
```

• • •

```
std::vector<int> resVec{1, 2, 3, 4, 5, 6, 7, 8};
```

```
std::exclusive_scan(std::execution::par,
```

```
resVec.begin(), resVec.end(), resVec.begin(), 1,
```

```
[(int fir, int sec){ return fir * sec; }];
```

```
for (auto v: resVec) std::cout << v << " "; // 1 1 2 6 24 120 720 5040
```

```
std::vector<int> resVec2{1, 2, 3, 4, 5, 6, 7, 8};
```

```
std::inclusive_scan(std::execution::par,
```

```
resVec2.begin(), resVec2.end(), resVec2.begin(),
```

```
[(int fir, int sec){ return fir * sec; }, 1);
```

```
for (auto v: resVec2) std::cout << v << " "; // 1 2 6 24 120 720 5040 40320
```

```
std::vector<int> resVec3{1, 2, 3, 4, 5, 6, 7, 8};
```

```
std::vector<int> resVec4(resVec3.size());
```

```
std::transform exclusive scan(std::execution::par,
```

```
resVec3.begin(), resVec3.end(),
```

```
resVec4.begin(), 0,
```

```
[(int fir, int sec){ return fir + sec; },
```

```
[(int arg){ return arg *= arg; }];
```

```
for (auto v: resVec4) std::cout << v << " "; // 0 1 5 14 30 55 91 140
```

```
std::vector<std::string> strVec{"Only", "for", "testing", "purpose"};
```

```
std::vector<int> resVec5(strVec.size()):
```

```
std::transform inclusive scan(std::execution::par,
```

```
strVec.begin(), strVec.end(),
```

```

        resVec5.begin(),
        [](auto fir, auto sec){ return fir + sec; },
        [](auto s){ return s.length(); }, 0);

for (auto v: resVec5) std::cout << v << " "; // 4 7 14 21

std::vector<std::string> strVec2{"Only", "for", "testing", "purpose"};

std::string res = std::reduce(std::execution::par,
                             strVec2.begin() + 1, strVec2.end(), strVec2[0],
                             [](auto fir, auto sec){ return fir + ":" + sec; });

std::cout << res;      // Only:for:testing:purpose

std::size_t res7 = std::transform_reduce(std::execution::par,
                                          strVec2.begin(), strVec2.end(), 0,
                                          [](std::size_t a, std::size_t b){ return a + b; },
                                          [](std::string s){ return s.length(); });

std::cout << res7;      // 21

```

Неинициализированная память

Следующие ниже функции в заголовке `<memory>` работают с неинициализированной памятью.

Скопировать диапазон объектов в неинициализированную область памяти:

```

FwdIt uninitialized_copy(InIt first, InIt last, FwdIt out)
FwdIt uninitialized_copy(ExePol pol, FwdIt first, FwdIt last, FwdIt out)

```

Скопировать count объектов в неинициализированную область памяти:

```

FwdIt uninitialized_copy_n(InIt first, Size count, FwdIt out)
FwdIt uninitialized_copy_n(ExePol pol, FwdIt first, Size count, FwdIt out)

```

Скопировать объект value в неинициализированную область памяти:

```

void uninitialized_fill(FwdIt first, FwdIt last, const T& value)
void uninitialized_fill(ExePol pol, FwdIt first, FwdIt last, const T& value)

```

Скопировать count объектов в неинициализированную область памяти начиная с first:

```

void uninitialized_fill_n(FwdIt first, Size count, const T& value)
FwdIt uninitialized_fill_n(FwdIt first, Size count, const T& value)
FwdIt uninitialized_fill_n(ExePol pol, FwdIt first, Size count, const T& value)

```

Переместить диапазон объектов в неинициализированную область памяти:

```
FwdIt uninitialized_move(InIt first, InIt last, FwdIt out)
FwdIt uninitialized_move(ExePol pol, FwdIt first, FwdIt last, FwdIt out)
```

Переместить count объектов в неинициализированную область памяти:

```
std::pair<InIt, FwdIt> uninitialized_move_n(InIt first, Size count, FwdIt out)
std::pair<FwdIt, FwdIt> uninitialized_move_n(ExePol pol, FwdIt first, Size count,
                                             FwdIt out)
```

Сконструировать объекты путем дефолтной инициализации¹ в неинициализированной области памяти:

```
void uninitialized_default_construct(FwdIt first, FwdIt last)
void uninitialized_default_construct(ExePol pol, FwdIt first, FwdIt last)
```

Сконструировать n объектов путем дефолтной инициализации в неинициализированной области памяти:

```
FwdIt uninitialized_default_construct_n(FwdIt first, Size n)
FwdIt uninitialized_default_construct_n(ExePol pol, FwdIt first, Size n)
```

Сконструировать объекты путем инициализации значениями² в неинициализированной области памяти:

```
void uninitialized_value_construct(FwdIt first, FwdIt last)
void uninitialized_value_construct(ExePol pol, FwdIt first, FwdIt last)
```

Сконструировать n объектов путем инициализации значениями в неинициализированной области памяти:

```
FwdIt uninitialized_value_construct_n(FwdIt first, Size n)
FwdIt uninitialized_value_construct_n(ExePol pol, FwdIt first, Size n)
```

Уничтожить объекты:

```
constexpr void destroy(FwdIt first, FwdIt last)
void destroy(ExePol pol, FwdIt first, FwdIt last)
```

Уничтожить n объектов:

```
constexpr void destroy_n(FwdIt first, Size n)
void destroy_n(ExePol pol, FwdIt first, Size n)
```

Уничтожить объекты по заданному адресу p:

```
constexpr void destroy_at(T* p)
```

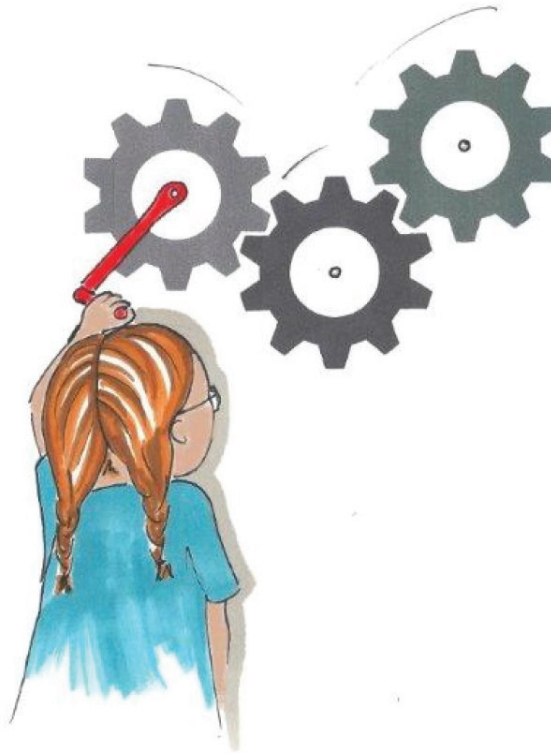
Создать объект по заданному адресу p:

```
constexpr T* construct_at(T* p, Args&& ... args)
```

¹ См. https://en.cppreference.com/w/cpp/language/default_initialization.

² См. https://en.cppreference.com/w/cpp/language/value_initialization.

11. Диапазоны



Циппи запускает работу конвейера

Краткий обзор

Библиотека `ranges` для работы с диапазонами была добавлена в стандарт C++20, но получила мощные расширения в стандарте C++23. Алгоритмы библиотеки `ranges` являются ленивыми, работают непосредственно на контейнере и допускают их сборку в композиции. Более того, большинство классических алгоритмов стандартной библиотеки шаблонов имеют диапазонные аналоги, которые поддерживают проекции и предоставляют дополнительные гарантии безопасности.

Композиция диапазонов

```
// rangesFilterTransform.cpp
...
#include <ranges>

std::vector<int> numbers = {1, 2, 3, 4, 5, 6};

auto results = numbers | std::views::filter([](int n){ return n % 2 == 0; }) //(4)
```

```
| std::views::transform([](int n){ return n * 2; });

for (auto v: results) std::cout << v << " ";    // 4 8 12
```

Приведенное в строке 4 выражение нужно читать слева направо. Символ вертикальной черты | обозначает композицию функций. Сначала пропускаются все четные числа (`std::views::filter([](int n){ return n % 2 == 0; })`). Затем каждое оставшееся число преобразовывается в удвоенное число (`std::views::transform([](int n){ return n * 2; })`).

Диапазон

Диапазоны и виды – это концепты¹. Концепт – это новая метапрограммная функциональность, введенная в стандарт C++20, которая позволяет задавать именованный набор условий или требований к аргументам шаблона. Стандарт C++20 поддерживает ряд разновидностей диапазонов.

std::ranges

Диапазон – это группа элементов, по которым можно выполнять итерации с помощью итераторов. В нем есть начальный итератор и конечный итератор. Концевые итераторы выполняют особую роль сторожков. Сторожки указывают на границы или конец диапазона в контейнерах или последовательностях. Помимо этого, у именованного пространства `std::ranges` имеются дополнительные тонкости.

Таблица. Тонкости именованного пространства `std::ranges`

Диапазон (именное пространство <code>std</code> опущено)	Описание
<code>ranges::input_range</code>	Задаёт диапазон, тип итератора которого удовлетворяет итератору ввода <code>input_iterator</code>
<code>ranges::output_range</code>	Задаёт диапазон, тип итератора которого удовлетворяет итератору вывода <code>output_iterator</code>
<code>ranges::forward_range</code>	Задаёт диапазон, тип итератора которого удовлетворяет вперёд-направленному итератору <code>forward_iterator</code>
<code>ranges::bidirectional_range</code>	Задаёт диапазон, тип итератора которого удовлетворяет двунаправленному итератору <code>bidirectional_iterator</code>
<code>ranges::random_access_range</code>	Задаёт диапазон, тип итератора которого удовлетворяет итератору с произвольным доступом <code>random_access_iterator</code>
<code>ranges::contiguous_range</code>	Задаёт диапазон, тип итератора которого удовлетворяет итератору со сплошным доступом <code>contiguous_iterator</code>

¹ См. <https://en.cppreference.com/w/cpp/language/constraints>.

Итератор с произвольным доступом `random_access_iterator` обеспечивает произвольный доступ к своим элементам и является неявным двунаправленным итератором `bidirectional_iterator`; двунаправленный итератор `bidirectional_iterator` позволяет выполнять итерации в обоих направлениях и является неявным впереднаправленным итератором `forward_iterator`; впереднаправленный итератор `forward_iterator` выполняет итерации в одном направлении. Интерфейс итератора со сплошным доступом `contiguous_iterator` аналогичен интерфейсу двунаправленного итератора `bidirectional_iterator`. Итератор со сплошным доступом `contiguous_iterator` гарантирует, что он обращается к сплошному участку хранилища, состоящему из смежно расположенных объектов.

Сторожок

Сторожок (sentinel) задает конец диапазона. Для контейнеров стандартной библиотеки шаблонов сторожком является концевой итератор. В стандарте C++20 тип сторожка может отличаться от типа начального итератора. В следующем ниже примере в качестве сторожка используется пробел.

Использование сторожка

```
// sentinelSpace.cpp
...
#include <algorithm>

...

struct Space {
    bool operator==(auto pos) const {
        return *pos == ' ';
    }
}

...

const char* rainerGrimm = "Rainer Grimm";
std::ranges::for_each(rainerGrimm, Space{}, [] (char c) { std::cout << c; });
// Rainer
```

Ввиду применения пробела в качестве сторожка последняя строка исходного кода выводит на консоль только строковый литерал `Rainer`.

`ranges::to`

В стандарте C++23 конкретная функция `std::ranges::to` представляет собой удобный способ конструирования контейнера из диапазона:

```
std::vector<int> range(int begin, int end, int stepsize = 1) {
    auto boundary = [end](int i){ return i < end; };
    std::vector<int> result = std::ranges::views::iota(begin)
```

```

        | std::views::stride(stepsize)
        | std::views::take_while(boundary)
        | std::ranges::to<std::vector>());
    return result;
}

```

Функция `range` создает контейнер `std::vector<int>`, состоящий из всех элементов от начала `begin` до конца `end` с размером шага `stepsize`. Величина аргумента `begin` должна быть меньше величины аргумента `end`.

Вид

Виды – это облегченные диапазоны. Вид позволяет обращаться к диапазонам, не копируя сами элементы, выполнять итерации по диапазонам, модифицировать или фильтровать элементы диапазона. Вид не владеет данными, а его временная сложность при копировании, перемещении или присваивании является постоянной.

Адаптеры диапазонов

Адаптер диапазона преобразовывает диапазон в вид.

```

std::vector<int> numbers = {1, 2, 3, 4, 5, 6};

auto results = numbers | std::views::filter([](int n){ return n % 2 == 0; })
                       | std::views::transform([](int n){ return n * 2; });

```

В приведенном выше фрагменте исходного кода `numbers` – это диапазон, а адаптеры `std::views::filter` и `std::views::transform` генерируют виды после применения соответствующих функций.

Библиотека `ranges` в стандарте C++20 имеет богатый набор видов.

Таблица. Виды в стандарте C++20

Вид	Описание
<code>std::views::all_t</code> <code>std::views::all</code>	Берет все элементы
<code>std::ranges::ref_view</code>	Берет все элементы другого диапазона
<code>std::ranges::filter_view</code> <code>std::views::filter</code>	Берет элементы, удовлетворяющие предикату
<code>std::ranges::transform_view</code> <code>std::views::transform</code>	Преобразовывает каждый элемент
<code>std::ranges::take_view</code> <code>std::views::take</code>	Берет первые <code>n</code> элементов другого вида
<code>std::ranges::take_while_view</code> <code>std::views::take_while</code>	Берет элементы другого вида до тех пор, пока предикат возвращает <code>true</code>

Вид	Описание
<code>std::ranges::drop_view</code> <code>std::views::drop</code>	Пропускает первые n элементов другого вида
<code>std::ranges::drop_while_view</code> <code>std::views::drop_while</code>	Пропускает начальные элементы другого вида до тех пор, пока предикат не вернет <code>false</code>
<code>std::ranges::join_view</code> <code>std::views::join</code>	Соединяет вид на диапазоны
<code>std::ranges::split_view</code> <code>std::views::split</code>	Разбивает вид с использованием разделителя
<code>std::ranges::common_view</code> <code>std::views::common</code>	Конвертирует вид в диапазон <code>std::ranges::common_range</code>
<code>std::ranges::reverse_view</code> <code>std::views::reverse</code>	Выполняет итерации в обратном порядке
<code>std::ranges::basic_istream_view</code> <code>std::ranges::istream_view</code>	Применяет оператор <code>>></code> к виду
<code>std::ranges::elements_view</code> <code>std::views::elements</code>	Создает вид на n -м элементе кортежей
<code>std::ranges::keys_view</code> <code>std::views::keys</code>	Создает вид на первом элементе парноподобных значений
<code>std::ranges::values_view</code> <code>std::views::values</code>	Создает вид на втором элементе парноподобных значений

В общем случае такой вид, как `std::views::transform`, можно использовать с альтернативным именем `std::ranges::transform_view`.

Благодаря библиотеке `ranges` алгоритмы могут применяться непосредственно к контейнерам, допускают их сборку в композиции и являются ленивыми.

Стандарт C++23 поддерживает дополнительные виды:

Таблица. Дополнительные виды в стандарте C++23

Вид	Описание
<code>std::ranges::zip_view</code> <code>std::views::zip</code>	Создает вид на кортежи
<code>std::ranges::zip_transform_view</code> <code>std::views::zip_transform</code>	Создает вид на кортежи с применением функции преобразования
<code>std::ranges::adjacent_view</code> <code>std::views::adjacent</code>	Создает вид на смежные элементы
<code>std::ranges::adjacent_transform_view</code>	Создает вид на смежные элементы с применением функции преобразования
<code>std::views::adjacent_transform</code> <code>std::ranges::join_with_view</code> <code>std::views::join_with</code>	Соединяет существующие диапазоны в вид с применением разделителя
<code>std::ranges::slide_view</code> <code>std::views::slide</code>	Создает n -элементные кортежи, беря вид и число n

Вид	Описание
<code>std::ranges::chunk_view</code> <code>std::views::chunk</code>	Создает n -элементные фрагменты вида с использованием числа n
<code>std::ranges::chunk_by_view</code> <code>std::views::chunk_by</code>	Создает фрагменты вида на основе предиката
<code>std::ranges::as_const_view</code> <code>std::views::as_const</code>	Конвертирует вид в константный диапазон
<code>std::ranges::as_rvalue_view</code> <code>std::views::as_rvalue</code>	Приводит каждый элемент в правостороннее значение
<code>std::ranges::stride_view</code> <code>std::views::stride</code>	Создает вид из n элементов другого вида

В следующем ниже фрагменте исходного кода применяются виды стандарта C++23.

Виды, введенные в стандарте C++23

```
// cpp23Ranges.cpp
...
#include <ranges>
...

std::vector vec = {1, 2, 3, 4};

for (auto i : vec | std::views::adjacent<2>) {
    std::cout << '(' << i.first << ", " << i.second << ") "; // (1, 2) (2, 3) (3, 4)
}

for (auto i : vec | std::views::adjacent_transform<2>(std::multiplies())) {
    std::cout << i << ' '; // 2 6 12
}

std::print("{}\n", vec | std::views::chunk(2)); // [[1, 2], [3, 4],
std::print("{}\n", vec | std::views::slide(2)); // [[1, 2], [2, 3], [3, 4]]

for (auto i : vec | std::views::slide(2)) {
    std::cout << '[' << i[0] << ", " << i[1] << "] "; // [1, 2] [2, 3] [3, 4] [4, 5]
}

std::vector vec2 = {1, 2, 3, 0, 5, 2};
std::print("{}\n", vec2 | std::views::chunk_by(std::ranges::less_equal{}));
// [[1, 2, 3], [0, 5], [2]]

for (auto i : vec | std::views::slide(2)) {
    std::cout << '[' << i[0] << ", " << i[1] << "] "; // [1, 2] [2, 3] [3, 4] [4, 5]
}
```

std::generator

Вспомогательный шаблонный (параметризуемый) класс `std::generator` в стандарте C++23 представляет собой первый конкретный генератор сопрограмм. Класс `std::generator` генерирует последовательность элементов путем многократного возобновления работы сопрограммы, в которой он был приостановлен.

`std::generator`

```
// generator.cpp
...
#include <generator>
#include <ranges>

...

std::generator<int> fib() {
    co_yield 0;                                // 1
    auto a = 0;
    auto b = 1;
    for(auto n : std::views::iota(0)) {
        auto next = a + b;
        a = b;
        b = next;
        co_yield next;                          // 2
    }
}

...

for (auto f : fib() | std::views::take(10)) { // 3
    std::cout << f << " ";                  // 0 1 1 2 3 5 8 13 21 34
}
```

Функция `fib` возвращает сопрограмму. Эта сопрограмма создает бесконечный поток чисел Фибоначчи. Поток чисел начинается с 0 (1) и продолжается следующим числом Фибоначчи (2). Диапазонный цикл `for` в явной форме запрашивает первые 10 чисел Фибоначчи (3).

Работа прямо на контейнерах

Алгоритмы стандартной библиотеки шаблонов нуждаются в начальном и конечном итераторах.

```
std::vector<int> myVec{1, 2, 3, 4, 5, 6, 7, 8, 9};
auto res = std::accumulate(std::begin(myVec), std::end(myVec), 0);
std::cout << res << '\n';                // 45
```

Библиотека `ranges` позволяет создавать вид прямо на ключах (1) либо значениях (3) контейнера `std::unordered_map`.

Диапазоны работают прямо на контейнере

```
// rangesEntireContainer.cpp
...

#include <ranges>

std::unordered_map<std::string, int> freqWord{ {"witch", 25}, {"wizard", 33},
                                              {"tale", 45}, {"dog", 4},
                                              {"cat", 34}, {"fish", 23} };

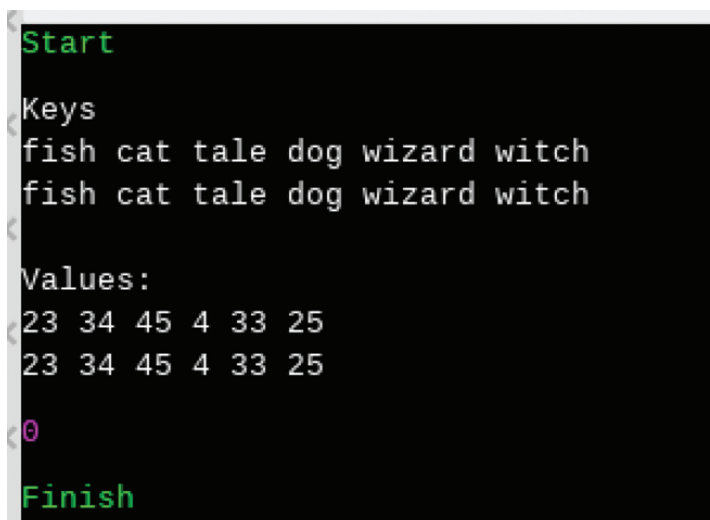
std::cout << "Keys" << '\n';
auto names = std::views::keys(freqWord); // (1)
for (const auto& name : names){ std::cout << name << " "; };

for (const auto& na : std::views::keys(freqWord)){ std::cout << na << " "; }; // (2)

std::cout << "Values: " << '\n';
auto values = std::views::values(freqWord); // (3)
for (const auto& value : values){ std::cout << value << " "; };

for (const auto& value : std::views::values(freqWord)){ // (4)
    std::cout << value << " ";
}
```

Разумеется, ключи и значения можно выводить на консоль напрямую ((2) и (4)). Результат будет идентичным.



```
Start
Keys
fish cat tale dog wizard witch
fish cat tale dog wizard witch

Values:
23 34 45 4 33 25
23 34 45 4 33 25

Finish
```

Композиция функций

Библиотека `ranges` поддерживает композицию функций с использованием символа вертикальной черты `|`.

Диапазоны с композицией функций

```
// rangesComposition.cpp
...

#include <ranges>

std::map<std::string, int> freqWord{ {"witch", 25}, {"wizard", 33}, {"tale", 45},
                                     {"dog", 4}, {"cat", 34}, {"fish", 23} };

std::cout << "All words: "; // (1)
for (const auto& name : std::views::keys(freqWord)) { std::cout << name << " "; } \

std::cout << "All words reverse: "; // (2)
for (const auto& name : std::views::keys(freqWord) | std::views::reverse) {
    std::cout << name << " ";
}

std::cout << "The first 4 words: "; // (3)
for (const auto& name : std::views::keys(freqWord) | std::views::take(4)) {
    std::cout << name << " ";
}

std::cout << "All words starting with w: "; // (4)
auto firstw = [](const std::string& name){ return name[0] == 'w'; };
for (const auto& name : std::views::keys(freqWord) | std::views::filter(firstw)) {
    std::cout << name << " ";
}
```

В данном случае интересуют только ключи. Сперва выводятся все ключи (1); затем все инвертированные ключи (2); потом первые четыре (3) и в конце ключи, начинающиеся с буквы 'w' (4).

Символ вертикальной черты `|` представляет собой синтаксический сахар, служащий для композиции функций. Вместо написания `C(R)` можно писать `R | C`. Следовательно, следующие ниже три строки эквивалентны.

```
auto rev1 = std::views::reverse(std::views::keys(freqWord));
auto rev2 = std::views::keys(freqWord) | std::views::reverse;
auto rev3 = freqWord | std::views::keys | std::views::reverse;
```

Наконец, ниже приведен результат работы программы:

```
Start
All words: cat dog fish tale witch wizard
All words reverse: wizard witch tale fish dog cat
The first 4 words: cat dog fish tale
All words starting with w: witch wizard
0
Finish
```

Ленивое оценивание

Генерирующий вид `std::views::iota`¹ – это фабрика диапазонов, служащая для создания последовательности элементов путем поочередного приращения начального значения. Эта последовательность может быть конечной либо бесконечной. Благодаря функциональности `std::views::iota` можно отыскать 20 первых простых чисел начиная с 1 000 000.

Отыскание первых 20 простых чисел начиная с 1 000 000

```
// rangesLazy.cpp
...

#include <ranges>

bool isPrime(int i) {
    for (int j=2; j*j <= i; ++j){
        if (i % j == 0) return false;
    }
    return true;
}

// (1)
std::cout << "Numbers from 1000000 to 1001000 (displayed each 100th): " << "\n";
for (int i: std::views::iota(1000000, 1001000)) {
    if (i % 100 == 0) std::cout << i << " ";
}

// (2)
auto odd = [](int i){ return i % 2 == 1; };
std::cout << "Odd numbers from 1000000 to 1001000 (displayed each 100th): " << "\n";
for (int i: std::views::iota(1000000, 1001000) | std::views::filter(odd)) {
    if (i % 100 == 1) std::cout << i << " ";
}

// (3)
std::cout << "Prime numbers from 1000000 to 1001000: " << '\n';
```

¹ См. https://en.cppreference.com/w/cpp/ranges/iota_view.

```

for (int i: std::views::iota(1000000, 1001000) | std::views::filter(odd)
                                     | std::views::filter(isPrime)) {
    std::cout << i << " ";
}

// (4)
std::cout << "20 prime numbers starting with 1000000: " << '\n';
for (int i: std::views::iota(1000000) | std::views::filter(odd)
                                     | std::views::filter(isPrime)
                                     | std::views::take(20)) {
    std::cout << i << " ";
}

```

Ниже приводится описание итерационной стратегии:

1. Нельзя сказать, когда будет получено 20 простых чисел, превышающих 1 000 000. Для подстраховки создается 1000 чисел. При этом на консоль выводится только каждое сотое число.
2. Простое число является нечетным, поэтому четные числа удаляются.
3. Предикат `isPrime` возвращает `true`, если число является простым. В результате получается 75 простых чисел, но нужно только 20.
4. В качестве фабрики бесконечных чисел, начиная с 1 000 000, используется вид `std::views::iota`. При этом запрашивается именно 20 простых чисел.

```

Numbers from 1000000 to 1001000 (displayed each 100th):
1000000 1000100 1000200 1000300 1000400 1000500 1000600 1000700 1000800 1000900

Odd numbers from 1000000 to 1001000 (displayed each 100th):
1000001 1000101 1000201 1000301 1000401 1000501 1000601 1000701 1000801 1000901

Prime numbers from 1000000 to 1001000:
1000003 1000033 1000037 1000039 1000081 1000099 1000117 1000121 1000133 1000151 1000159 1000171 1000183 1000187 1000193 1000199 1000211 1000213
1000231 1000249 1000253 1000273 1000289 1000291 1000303 1000313 1000333 1000357 1000367 1000381 1000393 1000397 1000403 1000409 1000423 1000427
1000429 1000453 1000457 1000507 1000537 1000541 1000547 1000577 1000579 1000589 1000609 1000619 1000621 1000639 1000651 1000667 1000669 1000679
1000691 1000697 1000721 1000723 1000763 1000777 1000793 1000829 1000847 1000849 1000859 1000861 1000889 1000907 1000919 1000921 1000931 1000969
1000973 1000981 1000999

20 prime numbers starting with 1000000:
1000003 1000033 1000037 1000039 1000081 1000099 1000117 1000121 1000133 1000151 1000159 1000171 1000183 1000187 1000193 1000199 1000211 1000213
1000231 1000249

```

Алгоритмы из std в сопоставлении с алгоритмами из std::ranges

Алгоритмы из библиотеки `algorithm`¹ и библиотеки `memory`² имеют диапазонные аналоги³. Они начинаются с именованного пространства `std::ranges`. Числовая библиотека `numeric`⁴ не имеет диапазонных аналогов. В следующем ниже листинге показана одна из пяти перегрузок стандартного алгоритма `std::sort` и одна из двух перегрузок нового алгоритма `std::ranges::sort`.

¹ См. <https://en.cppreference.com/w/cpp/header/algorithm>.

² См. <https://en.cppreference.com/w/cpp/header/memory>.

³ См. <https://en.cppreference.com/w/cpp/algorithm/ranges>.

⁴ См. <https://en.cppreference.com/w/cpp/header/numeric>.

```
template< class ExecutionPolicy, class RandomIt, class Compare >
void sort( ExecutionPolicy&& policy,
          RandomIt first, RandomIt last, Compare comp );

template <std::random_access_iterator I, std::sentinel_for<I> S,
          class Comp = ranges::less, class Proj = std::identity>
requires std::sortable<I, Comp, Proj>
constexpr I sort(I first, S last, Comp comp = {}, Proj proj = {});
```

Во-первых, стандартный алгоритм `std::sort` принимает диапазон, заданный начальным и конечным итераторами. Это должны быть итераторы с произвольным доступом. Они нуждаются в политике исполнения. Аргумент `Compare` позволяет конкретизировать стратегию сортировки в алгоритме `std::sort`.

Более того, если изучить перегрузку алгоритма `std::ranges::sort`, то можно заметить, что он принимает диапазон, заданный начальным итератором с произвольным доступом и сторожкой. Вдобавок эта перегрузка принимает предикат `Comp` и проекцию `Proj`. В предикате `Comp` по умолчанию используется операция `less`, а проекция `Proj` – это функциональный объект тождественного отображения `std::identity`¹. В данном случае проекция – это отображение множества в подмножество:

```
struct PhoneBookEntry{
    std::string name;
    int number;
};

std::vector<PhoneBookEntry> phoneBook{
    {"Brown", 111}, {"Smith", 444}, {"Grimm", 666},
    {"Butcher", 222}, {"Taylor", 555}, {"Wilson", 333} };

std::ranges::sort(phoneBook, std::ranges::greater(), &PhoneBookEntry::name);
```

Объект `phoneBook` сортируется в убывающем порядке на основе проекции `&PhoneBookEntry::name`.

`std::random_access_iterator`, `std::sortable` и `std::sentinel_for` – это концепты². Первый определяет параметры и поведение итераторов с произвольным доступом; второй определяет требования, которым должны соответствовать типы, чтобы их можно было сортировать с помощью алгоритмов сортировки, таких как `std::sort`; и третий используется для определения требований к типам, которые могут быть использованы в парах итераторов, чтобы гарантировать правильное поведение при выполнении итераций.

Алгоритм `std::ranges::sort` не поддерживает политики исполнения.

¹ См. <https://en.cppreference.com/w/cpp/utility/functional/identity>.

² См. <https://www.modernescpp.com/index.php/tag/concepts>.

12. Числа



Циппи учит арифметику

Краткий обзор

Язык C++ наследует числовые функции у языка C и имеет библиотеку `random` для работы со случайными числами¹.

Случайные числа

Случайные числа необходимы во многих областях применения, например для тестирования программного обеспечения, генерации криптографических ключей или для компьютерных игр. Библиотека C++ `random` состоит из двух компонентов. Это генерация случайных чисел и распределение случайных чисел. Обе части нуждаются в заголовке `<random>`.

Генератор случайных чисел

Генератор случайных чисел генерирует поток случайных чисел между минимальным и максимальным значениями. Этот поток инициализируется так называемой «затравкой» (*seed*), гарантирующей получение разных последовательностей случайных чисел.

```
#include <random>
```

```
...
```

```
std::random_device seed;  
std::mt19937 generator(seed());
```

¹ См. <http://en.cppreference.com/w/cpp/header/random>.

Генератор случайных чисел `gen` типа `Generator` поддерживает четыре разных запроса:

`Generator::result_type`

Возвращает тип данных сгенерированного случайного числа.

`gen()`

Возвращает случайное число.

`gen.min()`

Возвращает минимальное случайное число, которое может быть возвращено функцией `gen()`.

`gen.max()`

Возвращает максимальное случайное число, которое может быть возвращено функцией `gen()`.

Библиотека `random` поддерживает несколько генераторов случайных чисел. Наиболее известными являются зависящий от реализации вихрь Мерсенна (Mersenne Twister) `std::default_random_engine` и единственный настоящий генератор случайных чисел `std::random_device`, но не все платформы его поддерживают.

Распределение случайных чисел

Распределение случайных чисел соотносит случайное число, полученное с помощью генератора случайных чисел `gen`, с выбранным распределением.

```
#include <random>
```

```
...
```

```
std::random_device seed;
std::mt19937 gen(seed());
std::uniform_int_distribution<> unDis(0, 20); // распределение между 0 и 20
unDis(gen);                                // генерирует случайное число
```

В языке C++ есть несколько дискретных и непрерывных распределений случайных чисел. Дискретное распределение случайных чисел генерирует целые числа. Непрерывное распределение случайных чисел генерирует числа с плавающей точкой.

```
class bernoulli_distribution;
template<class T = int> class uniform_int_distribution;
template<class T = int> class binomial_distribution;
template<class T = int> class geometric_distribution;
template<class T = int> class negative_binomial_distribution;
template<class T = int> class poisson_distribution;
template<class T = int> class discrete_distribution;
```

```

template<class T = double> class exponential_distribution;
template<class T = double> class gamma_distribution;
template<class T = double> class weibull_distribution;
template<class T = double> class extreme_value_distribution;
template<class T = double> class normal_distribution;
template<class T = double> class lognormal_distribution;
template<class T = double> class chi_squared_distribution;
template<class T = double> class cauchy_distribution;
template<class T = double> class fisher_f_distribution;
template<class T = double> class student_t_distribution;
template<class T = double> class piecewise_constant_distribution;
template<class T = double> class piecewise_linear_distribution;
template<class T = double> class uniform_real_distribution;

```

Шаблонные классы с дефолтным аргументом типа `int` являются дискретными. Бернуллиево распределение генерирует булевы числа.

Ниже приведен пример использования вихря Мерсенна `std::mt19937` в качестве генератора псевдослучайных чисел, чтобы сгенерировать один миллион случайных чисел. Поток случайных чисел соотносится с равномерным и нормальным (или гауссовым) распределениями.

Случайные ЧИСЛА

```

// random.cpp
...
#include <random>

...

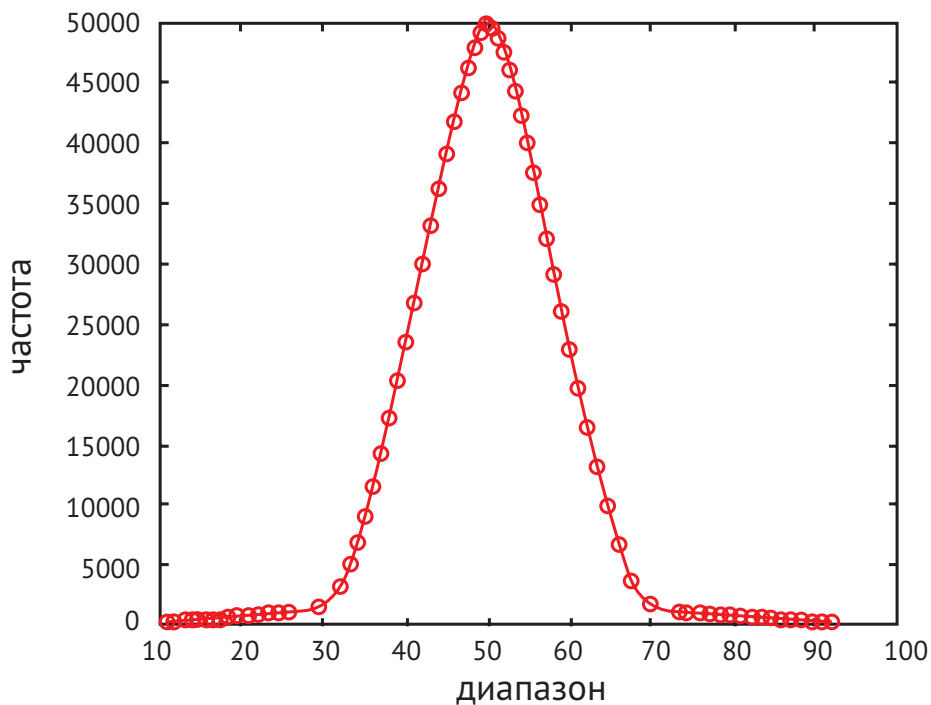
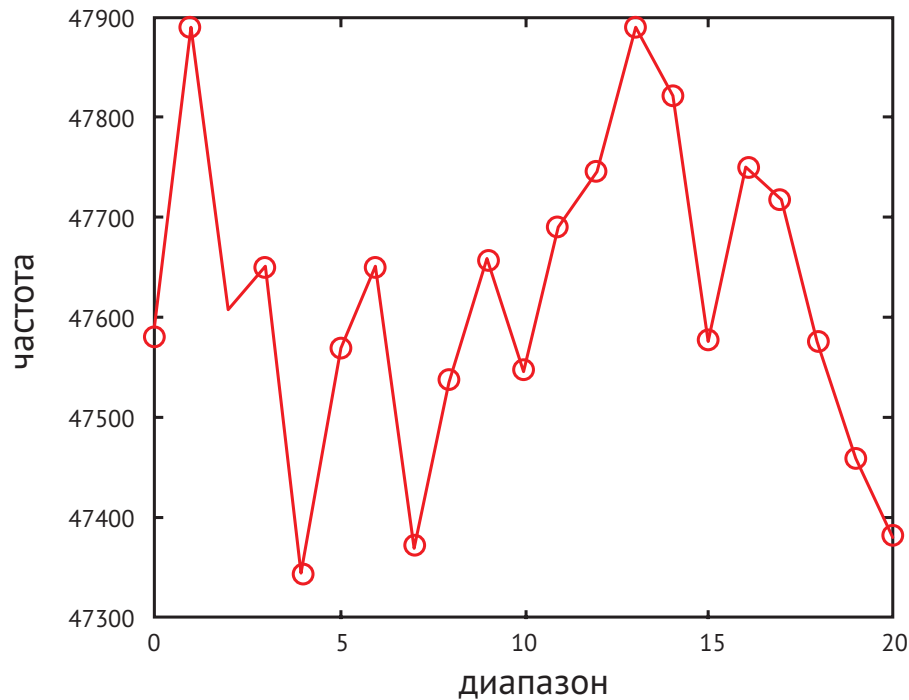
static const int NUM= 1000000;

std::random_device seed;
std::mt19937 gen(seed());
std::uniform_int_distribution<> uniformDist(0, 20); // мин= 0; макс= 20
std::normal_distribution<> normDist(50, 8);         // среднее= 50; сигма= 8

std::map<int, int> uniformFrequency;
std::map<int, int> normFrequency;
for (int i= 1; i <= NUM; ++i){
    ++uniformFrequency[uniformDist(gen)];
    ++normFrequency[round(normDist(gen))];
}

```

На следующих ниже графиках показаны равномерное и нормальное распределения миллиона случайных чисел.



Числовые функции, унаследованные у языка C

Язык C++ унаследовал у языка C целый ряд числовых функций. Для них необходим заголовок `<cmath>`¹. В приведенной ниже таблице показаны имена этих функций.

¹ См. <http://en.cppreference.com/w/cpp/numeric/math>.

Таблица. Математические функции из <cmath>

pow	sin	tanh	asinh	fabs
exp	cos	asin	aconsh	fmod
sqrt	tan	acos	atanh	frexp
log	sinh	atan	ceil	ldexp
log10	cosh	atan2	floor	modf

Дополнительно к этому язык C++ наследует у языка C несколько математических функций. Они определены в заголовке <cstdlib>¹. Опять-таки, ниже приведены их имена.

Таблица. Математические функции из <cstdlib>

abs	llabs	ldiv	srand
labs	div	lldiv	rand

Все целочисленные функции доступны для типов int, long и long long; все функции на числах с плавающей точкой доступны для типов float, double и 'long double.

Числовые функции должны квалифицироваться именным пространством std.

Математические функции

```
// mathFunctions.cpp
...
#include <cmath>
#include <cstdlib>

...

std::cout << std::pow(2, 10); // 1024
std::cout << std::pow(2, 0.5); // 1.41421
std::cout << std::exp(1); // 2.71828
std::cout << std::ceil(5.5); // 6
std::cout << std::floor(5.5); // 5
std::cout << std::fmod(5.5, 2); // 1.5

double intPart;
auto fracPart= std::modf(5.7, &intPart);
std::cout << intPart << " + " << fracPart; // 5 + 0.7
std::div_t divresult= std::div(14, 5);
std::cout << divresult.quot << " " << divresult.rem; // 2 4

// seed
```

¹ См. <http://en.cppreference.com/w/cpp/numeric/math>.

```
std::srand(time(nullptr));
for (int i= 0; i < 10; ++i) std::cout << (rand()%6 + 1) << " ";
// 3 6 5 3 6 5 6 3 1 5
```

Математические константы

Язык C++ поддерживает базовые и продвинутое математические константы. Математические константы имеют тип данных `double`. Они находятся в именном пространстве `std::number` и являются частью заголовка `<numbers>`.

Таблица. Математические константы

Математическая константа	Смысл
<code>std::numbers::e</code>	e
<code>std::numbers::log2e</code>	$\log_2 e$
<code>std::numbers::log10e</code>	$\log_{10} e$
<code>std::numbers::pi</code>	π
<code>std::numbers::inv_pi</code>	$1/\pi$
<code>std::numbers::inv_sqrtpi</code>	$1/\sqrt{\pi}$
<code>std::numbers::ln2</code>	$\ln 2$
<code>std::numbers::ln10</code>	$\ln 10$
<code>std::numbers::sqrt2</code>	$\sqrt{2}$
<code>std::numbers::sqrt3</code>	$\sqrt{3}$
<code>std::numbers::inv_sqrt3</code>	$1/\sqrt{3}$
<code>std::numbers::egamma</code>	Постоянная Эйлера–Маскерони ¹
<code>std::numbers::phi</code>	ϕ

Следующий ниже фрагмент исходного выводит на консоль все математические константы.

Математические константы

```
// mathematicalConstants.cpp

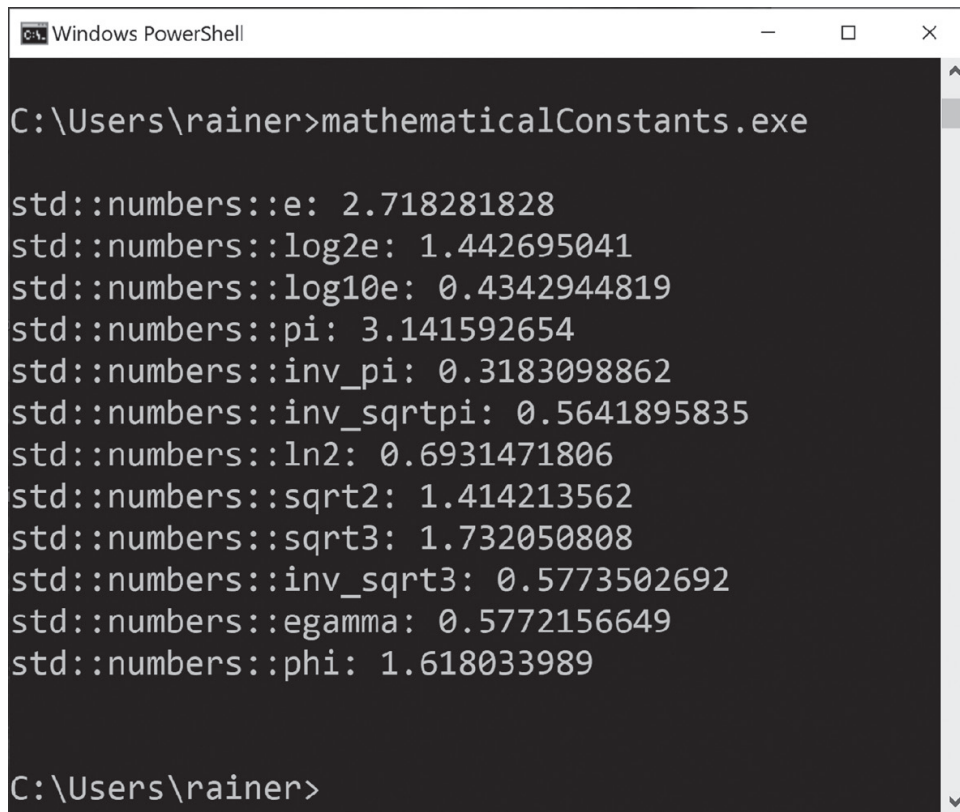
#include <numbers>

...

std::cout<< std::setprecision(10);
```

¹ См. https://en.wikipedia.org/wiki/Euler%E2%80%93Mascheroni_constant или https://en.wikipedia.org/wiki/Euler%27s_constant.

```
std::cout << "std::numbers::e: " << std::numbers::e << '\n';
std::cout << "std::numbers::log2e: " << std::numbers::log2e << '\n';
std::cout << "std::numbers::log10e: " << std::numbers::log10e << '\n';
std::cout << "std::numbers::pi: " << std::numbers::pi << '\n';
std::cout << "std::numbers::inv_pi: " << std::numbers::inv_pi << '\n';
std::cout << "std::numbers::inv_sqrtpi: " << std::numbers::inv_sqrtpi << '\n';
std::cout << "std::numbers::ln2: " << std::numbers::ln2 << '\n';
std::cout << "std::numbers::sqrt2: " << std::numbers::sqrt2 << '\n';
std::cout << "std::numbers::sqrt3: " << std::numbers::sqrt3 << '\n';
std::cout << "std::numbers::inv_sqrt3: " << std::numbers::inv_sqrt3 << '\n';
std::cout << "std::numbers::egamma: " << std::numbers::egamma << '\n';
std::cout << "std::numbers::phi: " << std::numbers::phi << '\n';
```



The screenshot shows a Windows PowerShell window with the title bar "C:\ Windows PowerShell". The command prompt shows the execution of "mathematicalConstants.exe" from the directory "C:\Users\rainer". The output displays various mathematical constants from the `std::numbers` namespace, each on a new line. The constants and their values are: `e` (2.718281828), `log2e` (1.442695041), `log10e` (0.4342944819), `pi` (3.141592654), `inv_pi` (0.3183098862), `inv_sqrtpi` (0.5641895835), `ln2` (0.6931471806), `sqrt2` (1.414213562), `sqrt3` (1.732050808), `inv_sqrt3` (0.5773502692), `egamma` (0.5772156649), and `phi` (1.618033989). The prompt returns to "C:\Users\rainer>" at the bottom.

```
C:\Users\rainer>mathematicalConstants.exe

std::numbers::e: 2.718281828
std::numbers::log2e: 1.442695041
std::numbers::log10e: 0.4342944819
std::numbers::pi: 3.141592654
std::numbers::inv_pi: 0.3183098862
std::numbers::inv_sqrtpi: 0.5641895835
std::numbers::ln2: 0.6931471806
std::numbers::sqrt2: 1.414213562
std::numbers::sqrt3: 1.732050808
std::numbers::inv_sqrt3: 0.5773502692
std::numbers::egamma: 0.5772156649
std::numbers::phi: 1.618033989

C:\Users\rainer>
```

13. Строковые объекты



Циппи играет в змейку

Краткий обзор

Базовый шаблонный класс `std::basic_string`¹ служит основой для более конкретных классов, таких как `std::string` и `std::wstring`, которые используют соответственно типы `char` и `wchar_t` и служат для работы с текстом. В частности, конкретный класс `std::string`, по сути дела, является контейнером, содержащим последовательность символов. В языке C++ большое число функций-членов предназначено для анализа или изменения содержимого строковых объектов. В частности, последовательности символов C++ (контейнеры `std::string`) являются безопасной заменой последовательностей символов C `const char*` (массивов символов, заканчивающихся символом `null`). Для работы со строковыми объектами необходим заголовок `<string>`.

M	y		S	t	r	i	n	g
---	---	--	---	---	---	---	---	---

¹ См. http://en.cppreference.com/w/cpp/string/basic_string.



Строковый объект `std::string` похож на контейнер `std::vector`

Строковый объект `std::string`, то есть контейнер последовательности смежно расположенных символов, похож на контейнер `std::vector`, содержащий символы. Он поддерживает очень похожий интерфейс, в силу чего для работы со строковым объектом доступны алгоритмы стандартной библиотеки шаблонов.

В следующем ниже фрагменте исходного кода строковый объект `std::string name` содержит строковый литерал `RainerGrimm`. Далее используется алгоритм стандартной библиотеки шаблонов `std::find_if`, чтобы получить заглавную букву, а затем извлекаются имя и фамилия, которые присваиваются строковым объектам `firstName` и `lastName`. Выражение `name.begin()+1` показывает, что строковые объекты поддерживают итераторы с произвольным доступом.

Строковый объект в сопоставлении с вектором

```
// string.cpp
...

#include <algorithm>
#include <string>

std::string name{"RainerGrimm"};
auto strIt= std::find_if(name.begin()+1, name.end(),
                        [](char c){ return std::isupper(c); });
if (strIt != name.end()){
    firstName= std::string(name.begin(), strIt);
    lastName= std::string(strIt, name.end());
}
```

Строковый объект реализован в виде шаблонного класса, параметризуемого символом, признаком символа (`traits`) и выделителем памяти (`Allocator`). Признак символа и выделитель имеют дефолтные значения, соответственно `char_traits<charT>` и `allocator<charT>`.

```
template <typename charT,
          typename traits= char_traits<charT>,
          typename Allocator= allocator<charT> >
class basic_string;
```

В языке C++ объявлены псевдонимы для символьных типов `char`, `wchar_t`, `char16_t` и `char32_t`.

```
typedef basic_string<char>          string;
typedef basic_string<wchar_t>      wstring;
typedef basic_string<char16_t>     u16string;
typedef basic_string<char32_t>     u32string;
```



Строковый объект `std::string` содержит строковый литерал

Когда программисты на языке C++ говорят о строковом объекте, то с вероятностью 99 % имеют в виду конкретизацию шаблонного класса `std::basic_string` символьным типом `char`. Это утверждение справедливо и для данной книги.

Создание и удаление

Язык C++ предлагает целый ряд функций-членов, служащих для создания строковых объектов из последовательностей символов C или C++. Под капотом при создании строкового объекта C++ всегда задействовалась последовательность символов C. Это изменилось с появлением нового стандарта C++14, поскольку он поддерживает последовательности символов C++: `std::string str{"string"s}`. Последовательность символов C "string literal" становится последовательностью символов C++ с суффиксом `s`: "string literal"s.

В приведенной ниже таблице показан краткий обзор функций-членов для создания и удаления строкового объекта.

Таблица. Функции-члены для создания и удаления строкового объекта

Функция-член	Пример
<code>std::string str</code>	Дефолтный конструктор
<code>std::string str(oth)</code>	Создание строкового объекта путем копирования из последовательности символов C++
<code>std::string str(std::move(oth))</code>	Создание строкового объекта путем перемещения из последовательности символов C++
<code>std::string(oth.begin(), oth.end())</code>	Создание строкового объекта из диапазона последовательности символов C++
<code>std::string(oth, otherIndex)</code>	Создание строкового объекта из подпоследовательности символов последовательности символов C++
<code>std::string(oth, otherIndex, strlen)</code>	Создание строкового объекта из подпоследовательности последовательности символов C++
<code>std::string str("c-string")</code>	Создание строкового объекта из последовательности символов C
<code>std::string str("c-array", len)</code>	Создание строкового объекта из массива C
<code>std::string str(num, 'c')</code>	Создание строкового объекта из символов
<code>std::string str({'a', 'b', 'c', 'd'})</code>	Создание строкового объекта из списка инициализации
<code>str= other.substring(3, 10)</code>	Создание строкового объекта из подпоследовательности символов
<code>str.~string()</code>	Деструктор

Создание строкового объекта

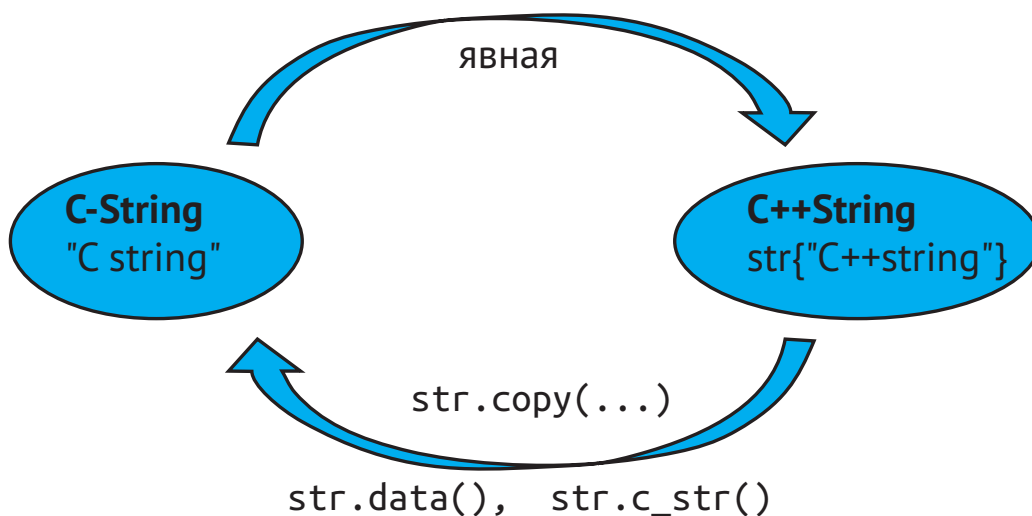
```
// stringConstructor.cpp
...
#include <string>

...

std::string defaultString;
std::string other{"123456789"};

std::string str1(other);           // 123456789
std::string tmp(other);           // 123456789
std::string str2(std::move(tmp));  // 123456789
std::string str3(other.begin(), other.end()); // 123456789
std::string str4(other, 2);        // 3456789
std::string str5(other, 2, 5);     // 34567
std::string str6("123456789", 5);  // 12345
std::string str7(5, '1');          // 11111
std::string str8({'1', '2', '3', '4', '5'}); // 12345
std::cout << str6.substr();        // 12345
std::cout << str6.substr(1);       // 2345
std::cout << str6.substr(1, 2);    // 23
```

Конвертация между последовательностями символов C++ и C



Конвертация последовательности символов C в последовательность символов C++ выполняется неявно, а конвертацию последовательности символов C++ в последовательность символов C необходимо запрашивать в явной форме.

ме. Функция-член `str.copy()` копирует последовательность символов C++ без завершающего символа `\0`. Функции-члены `str.data()` и `str.c_str()` включают завершающий символ `null`.



При работе с функциями `str.data()` и `str.c_str()` следует соблюдать осторожность

Значение, возвращаемое из двух функций-членов `str.data()` и `std.c_str()`, будет некорректным, если последовательность символов в строковом объекте `str` была модифицирована.

Последовательности символов C в сопоставлении с последовательностями символов C++

```
// stringCversusC++.cpp
...
#include<string>

...

std::string str{"C++-String"};

str += " C-String";

std::cout << str;    // строковый объект C++ - строковый объект C
const char* cString= str.c_str();
char buffer[10];
str.copy(buffer, 10);
str+= "works";
// const char* cString2= cString; // ОШИБКА
std::string str2(buffer, buffer+10);
std::cout<< str2;    // строковый объект C++
```

Размер в сопоставлении с емкостью

Количество элементов в строковом объекте (`str.size()`) в общем случае меньше, чем количество элементов, для которых зарезервировано место (`str.capacity()`), вследствие чего при добавлении элементов в строковый объект новая память автоматически не выделяется. Функция-член `std::max_size()` возвращает количество элементов, которое строковый объект может иметь максимально. Для трех функций-членов существует соотношение `str.size() <= str.capacity() <= str.max_size()`.

В следующей ниже таблице показаны функции-члены для управления памятью строкового объекта.

Таблица. Функции-члены для управления памятью строкового объекта

Функция-член	Описание
<code>str.empty()</code>	Проверяет существование элементов в строковом объекте <code>str</code>
<code>str.size()</code> , <code>str.length()</code>	Количество элементов в строковом объекте <code>str</code>
<code>str.capacity()</code>	Количество элементов, которое строковый объект <code>str</code> может иметь без перевыделения памяти
<code>str.max_size()</code>	Количество элементов, которое строковый объект <code>str</code> может иметь максимально
<code>str.resize(n)</code>	Изменение размера строкового литерала в строковом объекте <code>str</code> до <code>n</code> элементов
<code>str.resize_and_overwrite(n, op)</code>	Изменение размера строкового литерала в строковом объекте <code>str</code> до <code>n</code> элементов с применением операции <code>op</code> к его элементам
<code>str.reserve(n)</code>	Резервирование памяти для не менее чем <code>n</code> элементов
<code>str.shrink_to_fit()</code>	Подгонка емкости строкового объекта под его размер

Запрос `str.shrink_to_fit()`, как и в случае с контейнером `std::vector`, не является обязательным.

Размер в сопоставлении с емкостью

```
// stringSizeCapacity.cpp
...
#include <string>

...

void showStringInfo(const std::string& s){
    std::cout << s << ": ";
    std::cout << s.size() << " ";
    std::cout << s.capacity() << " ";
    std::cout << s.max_size() << " ";
}

std::string str;
showStringInfo(str); // "": 0 0 4611686018427387897

str += "12345";
showStringInfo(str); // "12345": 5 5 4611686018427387897

str.resize(30);
```

```
showStringInfo(str); // "12345": 30 30 4611686018427387897

str.reserve(1000);
showStringInfo(str); // "12345": 30 1000 4611686018427387897

str.shrink_to_fit();
showStringInfo(str); // "12345": 30 30 4611686018427387897
```

Сравнение

Строковые объекты поддерживают широко известные операторы сравнения `==`, `!=`, `<`, `>`, `>=`. Сравнение содержимого двух строковых объектов происходит поэлементно.

Сравнение строковых объектов

```
// stringComparisonAndConcatenation.cpp
...
#include <string>
...

std::string first{"aaa"};
std::string second{"aaaa"};

std::cout << (first < first) << '\n'; // false
std::cout << (first <= first) << '\n'; // true
std::cout << (first < second) << '\n'; // true
```

Конкатенация строковых объектов

Для строковых объектов используется перегруженный оператор `+`, поэтому строковые объекты можно складывать.



Оператор + перегружен только для последовательностей символов C++

Система типов C++ позволяет конкатенировать последовательности символов C++ и C с последовательностями символов C++, но не последовательности символов C++ и C с последовательностями символов C. Причина заключается в перегрузке оператора + для последовательностей символов C++, вследствие чего только вторая строка является правильно оформленным выражением языка C++, так как язык C выполняет неявное преобразование в последовательность символов C++:

Конкатенация строковых объектов

```
// stringComparisonAndConcatenation.cpp
...
#include <string>
...

std::string wrong= "1" + "1";           // ОШИБКА
std::string right= std::string("1") + "1"; // 11
```

Доступ к элементам

Доступ к элементам строкового объекта `str` очень удобен, так как конкретный класс `std::string` поддерживает итераторы с произвольным доступом. С помощью его функции-члена `str.front()` можно обращаться к первому символу строкового объекта, а с помощью функции-члена `str.back()` – к последнему символу. С помощью выражения `str[n]` и функции-члена `str.at(n)` можно получать n -й элемент по индексу.

В следующей ниже таблице представлен краткий обзор.

Таблица. Доступ к элементам строкового объекта

Функция-член	Пример
<code>str.front()</code>	Возвращает первый символ строкового объекта <code>str</code>
<code>str.back()</code>	Возвращает последний символ строкового объекта <code>str</code>
<code>str[n]</code>	Возвращает n -й символ строкового объекта <code>str</code> . Границы строкового объекта не проверяются
<code>str.at(n)</code>	Возвращает n -й символ строкового объекта <code>str</code> . Границы строкового объекта проверяются. При нарушении границ генерируется исключение <code>std::out_of_range</code>

Доступ к элементам строкового объекта

```
// stringAccess.cpp
...
#include <string>

...

std::string str= {"0123456789"};

std::cout << str.front() << '\n'; // 0
std::cout << str.back() << '\n';  // 9

for (int i= 0; i <= 3; ++i){
    std::cout << "str[" << i << "]: " << str[i] << "; ";
} // str[0]: 0; str[1]: 1; str[2]: 2; str[3]: 3;

std::cout << str[10] << '\n'; // неопределенное поведение
try{
    str.at(10);
}
catch (const std::out_of_range& e){
    std::cerr << "Exception: " << e.what() << '\n';
} // Exception: basic_string::at

std::cout << *(&str[0]+5) << '\n'; // 5
std::cout << *(&str[5]) << '\n';   // 5
std::cout << str[5] << '\n';       // 5
```

В приведенном выше примере особенно интересно отметить, как компилятор выполняет вызов `str[10]`. При доступе за пределы границ строкового объекта поведение программы становится неопределенным. С другой стороны, при вызове `str.at(10)` компилятор выражает протест.

Ввод и вывод данных

Строковый объект можно читать из потока ввода данных с помощью оператора `>>`, и его можно писать в поток вывода данных с помощью оператора `<<`.

Функция стандартной библиотеки `getline` позволяет читать из потока ввода данных построчно вплоть до символа конца файла.

У функции `getline` есть четыре вариации. Первыми двумя аргументами являются поток ввода данных `is` и строковый объект `line`, содержащий прочитанную строку текста. Опционально можно указывать специальный разделитель текстовых строк. Данная функция возвращает ссылку на поток ввода.

```
istream& getline (istream& is, string& line, char delim);
istream& getline (istream&& is, string& line, char delim);
istream& getline (istream& is, string& line);
istream& getline (istream&& is, string& line);
```

Функция `getline` потребляет всю текстовую строку, включая пустые пробелы. Игнорируется только разделитель текстовых строк. Данная функция нуждается в заголовке `<string>`.

Ввод и вывод строковых объектов

```
// stringInputOutput.cpp
...
#include <string>

...

std::vector<std::string> readFromFile(const char* fileName){
    std::ifstream file(fileName);
    if (!file){
        std::cerr << "Could not open the file " << fileName << ".";
        exit(EXIT_FAILURE);
    }
    std::vector<std::string> lines;
    std::string line;
    while (getline(file , line)) lines.push_back(line);
    return lines;
}

std::string fileName;
std::cout << "Your filename: ";
std::cin >> fileName;
std::vector<std::string> lines= readFromFile(fileName.c_str());
int num{0};
for (auto line: lines) std::cout << ++num << ": " << line << '\n';
```

Программа выводит на консоль строки произвольного файла, включая номер строки. Выражение `std::cin >> fileName` читает имя файла. Функция `readFromFile` читает с помощью функции `getline` все текстовые строки файла и помещает их в вектор.

Поиск

Язык C++ располагает возможностью поиска в строковом объекте в самых разных вариациях. Каждая вариация существует в различных перегруженных формах.



Имена алгоритмов поиска начинаются с префикса `find`

Как ни странно, имена алгоритмов поиска в строковом объекте начинаются с префикса `find`. Если поиск был успешным, то будет получен индекс типа `std::string::size_type`, который используется для представления размера строкового объекта. Если нет, то будет получена константа `std::string::npos`. Первый символ имеет индекс 0.

Алгоритмы поиска поддерживают:

- поиск символа, последовательности символов `C` либо последовательности символов `C++`;
- поиск символа и последовательности символов `C` либо последовательности символов `C++`;
- поиск в прямом направлении и в обратном направлении;
- положительный поиск (содержит) или отрицательный поиск (не содержит) символов из последовательности символов `C` либо последовательности символов `C++`;
- запуск поиска в строковом объекте с произвольной позиции.

Аргументы всех шести вариаций функции поиска имеют схожую структуру. Первым аргументом является искомый текст. Вторым аргументом – начальная позиция поиска, а третьим – количество символов, начиная со второго аргумента.

Ниже приведено шесть вариаций функции поиска.

Функция-член	Описание
<code>str.find(...)</code>	Возвращает первую позицию символа, последовательности символов <code>C</code> либо <code>C++</code> в строковом объекте <code>str</code>
<code>str.rfind(...)</code>	Возвращает последнюю позицию символа, последовательности символов <code>C</code> либо <code>C++</code> в строковом объекте <code>str</code>
<code>str.find_first_of(...)</code>	Возвращает первую позицию символа из последовательности символов <code>C</code> либо <code>C++</code> в строковом объекте <code>str</code>
<code>str.find_last_of(...)</code>	Возвращает последнюю позицию символа из последовательности символов <code>C</code> либо <code>C++</code> в строковом объекте <code>str</code>
<code>str.find_first_not_of(...)</code>	Возвращает первую позицию символа в строковом объекте <code>str</code> , который не является символом из последовательности символов <code>C</code> либо <code>C++</code>
<code>str.find_last_not_of(...)</code>	Возвращает последнюю позицию символа в строковом объекте <code>str</code> , который не является символом из последовательности символов <code>C</code> либо <code>C++</code>

Поиск в строковом объекте

```

// stringFind.cpp
...
#include <string>

...

std::string str;

auto idx= str.find("no");
if (idx == std::string::npos) std::cout << "not found"; // not found

str= {"dkeu84kf8k48kdj39kdj74945du942"};
std::string str2{"84"};

std::cout << str.find('8');           // 4
std::cout << str.rfind('8');          // 11
std::cout << str.find('8', 10);       // 11
std::cout << str.find(str2);          // 4
std::cout << str.rfind(str2);         // 4
std::cout << str.find(str2, 10);      // 18446744073709551615

str2="0123456789";

std::cout << str.find_first_of("678"); // 4
std::cout << str.find_last_of("678");  // 20
std::cout << str.find_first_of("678", 10); // 11
std::cout << str.find_first_of(str2);   // 4
std::cout << str.find_last_of(str2);    // 29
std::cout << str.find_first_of(str2, 10); // 10
std::cout << str.find_first_not_of("678"); // 0
std::cout << str.find_last_not_of("678"); // 29
std::cout << str.find_first_not_of("678", 10); // 10
std::cout << str.find_first_not_of(str2); // 0
std::cout << str.find_last_not_of(str2); // 26
std::cout << str.find_first_not_of(str2, 10); // 12

```

Вызов вспомогательной функции `std::find(str2, 10)` возвращает константу `std::string::npos`. При выводе этого значения на консоль в зависимости от платформы будет получено что-то вроде 18446744073709551615.

Проверка на наличие подпоследовательности символов

В стандарте C++20 функции-члены `str.starts_with(prefix)` и `str.ends_with(suffix)` проверяют, начинается ли содержимое строкового объекта `str` с префикса либо заканчивается суффиксом. Вдобавок в стандарте C++23 с помощью функции-члена `str.contains` можно выполнить проверку на существование подпоследовательности символов в строковом объекте.

Проверка на наличие префикса или суффикса

Функции-члены `str.starts_with(prefix)` и `str.ends_with(suffix)` проверяют, начинается ли содержимое заданного строкового объекта `str` с префикса либо заканчивается суффиксом. Подпоследовательность символов может быть видом `std::string_view`, одним символом либо строковым литералом.

Проверка на наличие у строкового объекта префикса либо суффикса

```
// startWithEndsWith.cpp
...
#include <string>

...

std::string helloWorld = "hello world";

std::cout << helloWorld.starts_with("hello") << '\n'; // true
std::cout << helloWorld.starts_with("llo") << '\n';   // false

std::cout << helloWorld.ends_with("world") << '\n';   // true
std::cout << helloWorld.ends_with("wo") << '\n';      // false
```

Проверка на наличие содержащейся подпоследовательности символов

Функция-член `str.contains` выполняет проверку на существование подпоследовательности символов в строковом объекте. Подпоследовательность символов может быть видом `std::string_view`, одним символом либо строковым литералом.

Проверка на наличие подпоследовательности символов в строковом объекте

```
// containsString.cpp
...
#include <string>
```

...

```
std::string helloWorld = "hello world";

std::cout << helloWorld.contains("hello") << '\n'; // true
std::cout << helloWorld.contains("llo") << '\n';   // true
std::cout << helloWorld.contains('w') << '\n';     // true
std::cout << helloWorld.contains('W') << '\n';     // false
```

Модифицирующие операции

Строковые объекты имеют целый ряд операций по их изменению. Функция-член `str.assign` назначает новый строковый литерал строковому объекту `str`. С помощью функции-члена `str.swap` можно менять два строковых объекта местами. Для удаления символа из строкового объекта используются функции-члены `str.pop_back` либо `str.erase`. Напротив, функции-члены `str.clear` и `str.erase` стирают все содержимое строкового объекта. Для добавления новых символов в строковый объект используется оператор `+=` либо функции-члены `std.append` и `str.push_back`. Для вставки новых символов можно использовать функцию-член `str.insert`, а для замены символов – функцию-член `str.replace`.

Таблица. Функции-члены для модификации строкового объекта

Функция-член	Описание
<code>str = str2</code>	Присваивает строковому объекту <code>str</code> строковый объект <code>str2</code>
<code>str.assign(...)</code>	Назначает строковому объекту <code>str</code> новый строковый литерал
<code>str.swap(str2)</code>	Меняет местами строковые объекты <code>str</code> и <code>str2</code>
<code>str.pop_back()</code>	Удаляет последний символ из строкового объекта <code>str</code>
<code>str.erase(...)</code>	Удаляет символы из строкового объекта <code>str</code>
<code>str.clear()</code>	Очищает строковый объект <code>str</code>
<code>str.append(...)</code>	Добавляет символы в строковый объект <code>str</code>
<code>str.push_back(s)</code>	Добавляет символ <code>s</code> в строковый объект <code>str</code>
<code>str.insert(pos, ...)</code>	Вставляет символы в строковый объект <code>str</code> , начиная с позиции <code>pos</code>
<code>str.replace(pos, len, ...)</code>	Заменяет <code>len</code> символов в строковом объекте <code>str</code> , начиная с позиции <code>pos</code>

Указанные операции доступны в многочисленных перегруженных версиях. Функции-члены `str.assign`, `str.append`, `str.insert` и `str.replace` очень похожи. Все четыре можно вызывать со строковыми объектами и подпоследовательностями символов C++, символами, последовательностями символов C, мас-

сивами последовательностей символов C, диапазонами и списками инициализации. Функция-член `str.erase` может стирать один символ, диапазоны и несколько символов, начиная с заданной позиции.

В следующем ниже фрагменте исходного кода продемонстрированы почти все вариации. Ради простоты показаны только эффекты модификации строковых объектов.

Модификация строковых объектов

```
// stringModification.cpp
...
#include <string>
...

std::string str{"New String"};
std::string str2{"Other String"};

str.assign(str2, 4, std::string::npos); // r String
str.assign(5, '-');                     // -----

str= {"0123456789"};
str.erase(7, 2); // 01234569
str.erase(str.begin()+2, str.end()-2); // 012
str.erase(str.begin()+2, str.end());   // 01
str.pop_back();                        // 0
str.erase();                           //

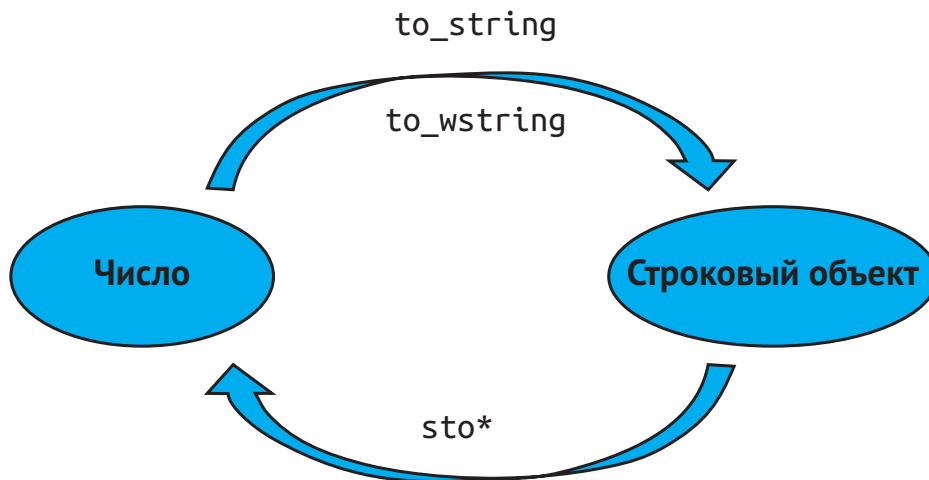
str= "01234";
str+= "56";                            // 0123456
str+= '7';                             // 01234567
str+= {'8', '9'};                      // 0123456789
str.append(str);                       // 01234567890123456789
str.append(str, 2, 4);                 // 012345678901234567892345
str.append(3, '0');                   // 012345678901234567892345000
str.append(str, 10, 10);               // 01234567890123456789234500001234567989
str.push_back('9');                   // 012345678901234567892345000012345679899

str= {"345"};
str.insert(3, "6789");                 // 3456789
str.insert(0, "012");                  // 0123456789

str= {"only for testing purpose."};
str.replace(0, 0, "0");                 // Only for testing purpose.
str.replace(0, 5, "Only", 0, 4);        // Only for testing purpose.
str.replace(16, 8, "");                 // Only for testing.
str.replace(4, 0, 5, 'y');              // Onlyyyyyy for testing.
```

```
str.replace(str.begin(), str.end(), "Only for testing purpose.");
// Only for testing purpose.
str.replace(str.begin()+4, str.end()-8, 10, '#');
// Only#####purpose.
```

Числовые конвертации



С помощью функций-членов `std::to_string(val)` и `std::to_wstring(val)` можно конвертировать числа или числа с плавающей точкой в соответствующие строковые объекты конкретных классов `std::string` и `std::wstring`. Для конвертации строковых объектов в обратном направлении в числа или числа с плавающей точкой существует семейство функций `sto*`. Все функции нуждаются в заголовке `<string>`.



Функции-члены `sto*` читаются как «string to» (строковый объект в)

Все семь способов конвертации строкового объекта в натуральное число или число с плавающей точкой подчиняются простой схеме. Все функции начинаются с имени `sto`, за которым следуют другие символы, обозначающие тип, в который нужно конвертировать строковый объект. Например, `stol` означает конвертацию строкового объекта в тип `long`, а `stod` – конвертацию строкового объекта в тип `double`.

Все функции `sto` имеют одинаковый интерфейс. В приведенном ниже примере показана функция для типа `long`.

```
std::stol(str, idx= nullptr, base= 10)
```

Указанная функция принимает строковый объект и определяет представление с типом `long` с основанием `base`. Функция `stol` игнорирует начальные

пробелы и опционально возвращает индекс первого недопустимого символа в `idx`. По умолчанию основание равно 10. Допустимыми значениями основания являются 0 и 2 до 36. При использовании основания 0 компилятор определяет тип автоматически, основываясь на формате содержимого строкового объекта. Если основание больше десяти, то компилятор кодирует их в символах от `a` до `z`. Представление аналогично представлению шестнадцатеричных чисел.

В приведенной ниже таблице дан краткий обзор всех функций-членов.

Таблица. Числовая конвертация содержимого строковых объектов

Функция-член	Описание
<code>std::to_string(val)</code>	Конвертирует значение <code>val</code> в <code>std::string</code>
<code>std::to_wstring(val)</code>	Конвертирует значение <code>val</code> в <code>std::wstring</code>
<code>std::stoi(str)</code>	Возвращает значение типа <code>int</code>
<code>std::stol(str)</code>	Возвращает значение типа <code>long</code>
<code>std::stoll(str)</code>	Возвращает значение типа <code>long long</code>
<code>std::stoul(str)</code>	Возвращает значение типа <code>unsigned long</code>
<code>std::stoull(str)</code>	Возвращает беззнаковое значение типа <code>long long</code>
<code>std::stof(str)</code>	Возвращает значение типа <code>float</code>
<code>std::stod(str)</code>	Возвращает значение типа <code>double</code>
<code>std::stold(str)</code>	Возвращает значение <code>long double</code>



Где находится функция `stou`?

Для справки, функции C++ `sto` являются тонкими обертками вокруг функций C `strto`. Однако в языке C нет функции `strto`, и поэтому в языке C++ нет функции `sto`.

В случае невозможности конвертации все функции генерируют исключение `std::invalid_argument`. Если определенное значение слишком велико для целевого типа, то генерируется исключение `std::out_of_range`.

Числовая конвертация

```
// stringNumericConversion.cpp
...
#include <string>
...

std::string maxLongLongString=
    std::to_string(std::numeric_limits<long long>::max());
```

```
std::wstring maxLongLongWstring=
    std::to_wstring(std::numeric_limits<long long>::max());

std::cout << std::numeric_limits<long long>::max(); // 9223372036854775807
std::cout << maxLongLongString; // 9223372036854775807
std::wcout << maxLongLongWstring; // 9223372036854775807

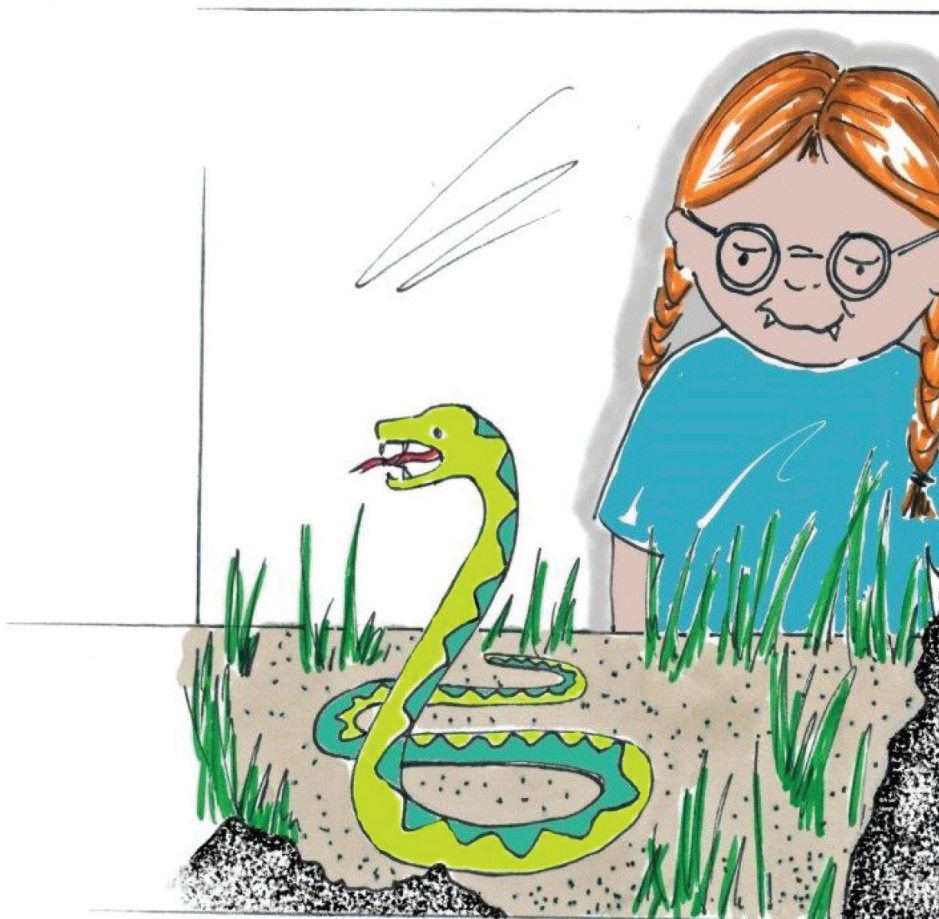
std::string str("10010101");

std::cout << std::stoi(str); // 10010101
std::cout << std::stoi(str, nullptr, 16); // 268501249
std::cout << std::stoi(str, nullptr, 8); // 2101313
std::cout << std::stoi(str, nullptr, 2); // 149

std::size_t idx;
std::cout << std::stod(" 3.5 km", &idx); // 3.5
std::cout << idx; // 6

try{
    std::cout << std::stoi(" 3.5 km") << '\n'; // 3
    std::cout << std::stoi(" 3.5 km", nullptr, 2) << '\n';
}
catch (const std::exception& e){
    std::cerr << e.what() << '\n';
} // stoi
```

14. Виды на строковые объекты



Циппи наблюдает за змеей

Краткий обзор

Вид на строковый объект^{1,2} представляет собой невладеющую ссылку на строковый объект. Он представляет собой вид на последовательность символов. Указанная последовательность может быть последовательностью символов C++ либо последовательностью символов C. Для получения функциональности обработки видов на строковые объекты необходим заголовок `<string_view>`.

¹ См. http://en.cppreference.com/w/cpp/string/basic_string_view.

² В качестве аналогии нередко приводится окно в доме и автомобиль возле окна. Можно посмотреть в окно и увидеть машину, но при этом невозможно дотронуться до машины или передвинуть ее. Окно обеспечивает лишь вид на автомобиль, который является отдельным независимым объектом. – *Прим. перев.*



Виды на строковые объекты предназначены для оптимизированных под копирование строковых объектов

С высоты птичьего полета цель объявления обертки `std::string_view` вокруг конкретного класса `std::string` состоит в том, чтобы избежать копирования данных, которые уже принадлежат кому-то другому, и обеспечивать немутулируемый доступ к строковому объекту класса `std::string`. Вид `std::string_view` является ограниченным строковым объектом, который поддерживает только немутулируемые операции. Вдобавок объект `std::string_view sv` имеет две дополнительные мутулирующие операции `sv.remove_prefix` и `sv.remove_suffix`.

Виды на строковые объекты – это шаблонные классы, параметризуемые символом и признаком символа. Признак символа имеет дефолтное значение. В отличие от строкового объекта, вид на строковый объект не является владельцем и, следовательно, не нуждается в выделителе памяти.

```
template<
    class CharT,
    class Traits = std::char_traits<CharT>
> class basic_string_view;
```

Как и в случае со строковыми объектами, виды на строковые объекты в C++ объявлены в качестве четырех псевдонимов для базисных символьных типов `char`, `wchar_t`, `char16_t` и `char32_t`.

```
typedef std::string_view      std::basic_string_view<char>
typedef std::wstring_view    std::basic_string_view<wchar_t>
typedef std::u16string_view  std::basic_string_view<char16_t>
typedef std::u32string_view  std::basic_string_view<char32_t>
```



Обертка `std::string_view` – это вид на строковый объект

Если говорить о виде на строковый объект, то в языке C++ с вероятностью 99 % имеется в виду конкретизация шаблонного класса `std::basic_string_view` для символьного типа `char`. Это утверждение справедливо и для данной книги.

Создание и инициализация

В языке C++ можно создавать пустой вид на строковый объект. Также можно создавать вид на строковый объект из существующего строкового объекта, массива символов либо вида на строковый объект.

В приведенной ниже таблице дан краткий обзор различных способов создания вида на строковый объект.

Таблица. Функции-члены для создания и установки вида на строковый объект

Функция-член	Описание
<code>std::string_view str_view</code>	Вид на пустой строковый объект
<code>std::string_view str_view2("C-string")</code>	Создание вида из последовательности символов C
<code>std::string_view str_view3(str_view2)</code>	Создание вида из вида на строковый объект
<code>std::string_view str_view4(arr, sizeof arr)</code>	Создание вида из массива C
<code>str_view4= str_view3.substr(2, 3)</code>	Создание вида из вида на строковый объект
<code>std::string_view str_view5 = str_view4</code>	Установка вида на строковый объект

Немодифицирующие операции

Ради краткости изложения в этой главе и для того, чтобы не повторять подробные описания из главы о строковых объектах, здесь будут упомянуты только немодифицирующие операции вида на строковый объект. Для получения более подробной информации следует воспользоваться ссылкой на соответствующую документацию в главе о строковых объектах.

- Доступ к элементам: оператор `[]` и функции-члены `at`, `front`, `back`, `data` (см. раздел «Доступ к элементам» главы о строковых объектах).
- Емкость: функции-члены `size`, `length`, `max_size`, `empty` (см. раздел «Размер в сопоставлении с емкостью» главы о строковых объектах).
- Поиск: функции-члены `find`, `rfind`, `find_first_of`, `find_last_of`, `find_first_not_of`, `find_last_not_of` (см. раздел «Поиск» главы о строковых объектах).
- Копирование: функция-член `copy` (см. раздел «Конвертация между последовательностями символов C++ и последовательностями символов C» главы о строковых объектах).

Модифицирующие операции

Вызов функции-члена `stringView.swap(stringView2)` меняет местами содержимое двух видов на строковые объекты. Функции-члены `remove_prefix` и `remove_suffix` являются для вида на строковый объект уникальными, так как строковый объект не поддерживает ни одну из них. Функция-член `remove_prefix` суживает содержимое строкового объекта с его начала в прямом направлении, а функция-член `remove_suffix` суживает содержимое строкового объекта с его конца в обратном направлении.

Модифицирующие и немодифицирующие операции

```
// string_view.cpp
```

```
...
```

```

#include <string_view>

...

using namespace std;

string str = " A lot of space";
string_view strView = str;

strView.remove_prefix(min(strView.find_first_not_of(" "), strView.size()));
cout << str << endl      // " A lot of space
    << strView << endl; // "A lot of space"

char arr[] = {'A', ' ', 'l', 'o', 't', ' ', 'o', 'f', ' ',
              's', 'p', 'a', 'c', 'e', '\0', '\0', '\0'};
string_view strView2(arr, sizeof arr);
auto trimPos = strView2.find('\0');
if(trimPos != strView2.npos) strView2.remove_suffix(strView2.size() - trimPos);
cout << arr << ": " << sizeof arr << endl      // A lot of space: 17
    << strView2 << ": " << strView2.size() << endl; // A lot of space: 14

```



При использовании вида на строковый объект выделение памяти не требуется

При создании или копировании вида на строковый объект выделять память не требуется. Это отличается от строкового объекта; операции создания или копирования строкового объекта нуждаются в выделении памяти.

Выделение памяти

```

// stringView.cpp
...
include <string_view>

...

void* operator new(std::size_t count){
    std::cout << " " << count << " bytes" << '\n';
    return malloc(count);
}

void getString(const std::string&){}

void getStringView(std::string_view){}

std::string large = "012345678901234567890"
                  "1234567890123456789"; // 41 bytes allocated

```

```
std::string substr = large.substr(10);           // 31 bytes allocated

std::string_view largeStringView{large.c_str(), // 0 bytes allocated
                                  large.size()};
largeStringView.remove_prefix(10);               // 0 bytes allocated

getString(large);
getString("012345678901234567890"
          "1234567890123456789"); // 41 bytes allocated
const char message []= "0123456789012345678901234567890123456789";
getString(message);                          // 41 bytes allocated

getStringView(large);                         // 0 bytes allocated
getStringView("012345678901234567890"
              "1234567890123456789");         // 0 bytes allocated
getStringView(message);                       // 0 bytes allocated
```

Благодаря глобальному перегруженному оператору `::operator new` можно наблюдать за каждым выделением памяти в куче.

15. Регулярные выражения



Циппи анализирует следы на снегу

Краткий обзор

Регулярные выражения¹ – это язык для описания текстовых шаблонов. Для работы с регулярными выражениями необходим заголовок `<regex>`.

Регулярные выражения являются мощным инструментом решения следующих ниже задач:

- проверка совпадения текста с текстовым шаблоном: шаблонная функция `std::regex_match`;
- поиск текстового шаблона в тексте: шаблонная функция `std::regex_search`;
- замена текстового шаблона текстом: шаблонная функция `std::regex_replace`;
- итеративный обход всех текстовых шаблонов в тексте: итераторные шаблонные классы `std::regex_iterator` и `std::regex_token_iterator`.

Язык C++ поддерживает шесть грамматик регулярных выражений. По умолчанию используется грамматика ECMAScript. Эта грамматика является самой мощной из шести грамматик и очень похожа на грамматику, используемую в языке Perl 5. Остальные пять грамматик – это базовая, расширенная, `awk`, `grep` и `egrep`.

¹ См. <http://en.cppreference.com/w/cpp/regex>.



Использование сырых строковых литералов

В регулярных выражениях следует использовать сырые строковые литералы. Регулярное выражение для текста «C++» выглядит довольно уродливо: `C\\+\\+`. Для каждого знака `+` нужно использовать две обратные косые черты. Во-первых, знак `+` играет уникальную роль в регулярном выражении. Во-вторых, обратная косая черта является специальным символом в строковом литерале. По этой причине одна обратная косая черта экранирует знак `+`, а другая – обратную косую черту. При использовании сырого строкового литерала вторая обратная косая черта больше не нужна, так как обратная косая черта в строковом литерале не интерпретируется.

```
#include <regex>
...
std::string regExpr("C\\+\\+");
std::string regExprRaw(R"(C\+\+)");
```

Работа с регулярными выражениями обычно выполняется в три этапа:

1. Определить регулярное выражение:

```
std::string text="C++ or c++.";
std::string regExpr(R"(C\+\+)");
std::regex rgx(regExpr);
```

2. Сохранить результат поиска:

```
std::smatch result;
std::regex_search(text, result, rgx);
```

3. Обработать результат:

```
std::cout << result[0] << '\n';
```

Типы символов

Тип текста обуславливает тип символов регулярного выражения и тип результата поиска.

В приведенной ниже таблице показаны четыре разные комбинации.

Таблица. Комбинации типа текста, регулярного выражения, результата поиска и действия

Тип текста	Тип регулярного выражения	Тип результата
<code>const char*</code>	<code>std::regex</code>	<code>std::cmatch</code>
<code>std::string</code>	<code>std::regex</code>	<code>std::smatch</code>
<code>const wchar_t*</code>	<code>std::wregex</code>	<code>std::wcmatch</code>
<code>std::wstring</code>	<code>std::wregex</code>	<code>std::wsmatch</code>

Программа, показанная в разделе «Поиск» этой главы, подробно описывает эти четыре комбинации.

Объекты регулярного выражения

Объекты регулярного выражения являются экземплярами шаблонного класса `template <class charT, class traits= regex_traits <charT>> class basic_regex`, параметризуемыми своим символьным типом и классом признаков. Класс признаков задает интерпретацию свойств грамматики регулярных выражений. В языке C++ объявлено два псевдонима базового шаблонного класса `basic_regex`:

```
typedef basic_regex<char>      regex;
typedef basic_regex<wchar_t>  wregex;
```

Объект регулярного выражения допускает дальнейшую настройку. Так, можно конкретизировать используемую грамматику или адаптировать синтаксис. Как уже упоминалось, язык C++ поддерживает базовую и расширенную грамматику, а также грамматики `awk`, `grep` и `egrep`. Регулярное выражение, квалифицированное флагом `std::regex_constants::icase`, нечувствительно к буквенному регистру. Если требуется адаптировать синтаксис, то необходимо конкретизировать грамматику явным образом.

Конкретизация грамматики

```
// regexGrammar.cpp
...
#include <regex>

...

using std::regex_constants::ECMAScript;
using std::regex_constants::icase;

std::string theQuestion="C++ or c++, that's the question.";
std::string regExprStr(R"(c\+\+)");

std::regex rgx(regExprStr);
std::smatch smatch;

if (std::regex_search(theQuestion, smatch, rgx)){
    std::cout << "case sensitive: " << smatch[0];    // c++
}

std::regex rgxIn(regExprStr, ECMAScript|icase);
if (std::regex_search(theQuestion, smatch, rgxIn)){
    std::cout << "case insensitive: " << smatch[0]; // C++
}
```

При использовании чувствительного к буквенному регистру регулярного выражения `rgx` результатом поиска в тексте строкового объекта `theQuestion` будет строковый литерал `c++`. Этого не происходит, если применяется нечувствительное к буквенному регистру регулярное выражение `rgxIn`. В этом случае при совпадении получится строковый литерал `C++`.

Результат поиска `match_results`

Входящий в стандартную библиотеку C++ шаблонный класс `std::match_results` призван хранить результаты операции установления совпадения, выполненной с помощью регулярных выражений. Объект класса `std::match_results` представляет результат работы шаблонных функций `std::regex_match` или `std::regex_search` и является контейнером последовательности символов, содержащим по меньшей мере одну группу захвата для объекта `std::sub_match`. Объект `std::sub_match` – это последовательности символов, представляющие конкретную со-
впавшую порцию регулярного выражения.



Что такое группа захвата?

Группы захвата позволяют выполнять дальнейший анализ результата поиска в регулярном выражении. Они определяются парой круглых скобок `( )`. Регулярное выражение `((a+)(b+)(c+))` имеет четыре группы захвата: `((a+)(b+)(c+))`, `(a+)`, `(b+)` и `(c+)`. Нулевая группа захвата является итоговым результатом.

В языке C++ объявлено четыре псевдонима шаблонного класса `std::match_results`:

```
typedef match_results<const char*>          smatch;
typedef match_results<const wchar_t*>       wcmatch;
typedef match_results<string::const_iterator> smatch;
typedef match_results<wstring::const_iterator> wsmatch;
```

Шаблонный класс `std::smatch` имеет мощный интерфейс.

Таблица. Интерфейс шаблонного класса `std::smatch` на примере объекта `smatch`

Функция-член	Описание
<code>smatch.size()</code>	Возвращает количество групп захвата
<code>smatch.empty()</code>	Определяет, есть ли внутри результата поиска группа захвата
<code>smatch[i]</code>	Возвращает <i>i</i> -ю группу захвата
<code>smatch.length(i)</code>	Возвращает длину <i>i</i> -й группы захвата
<code>smatch.position(i)</code>	Возвращает позицию <i>i</i> -й группы захвата
<code>smatch.str(i)</code>	Возвращает <i>i</i> -ю группу захвата в виде строкового объекта

<code>smatch.prefix()</code> и <code>smatch.suffix()</code>	Возвращает строковый литерал до и после группы захвата
<code>smatch.begin()</code> и <code>smatch.end()</code>	Возвращает начальный и конечной итераторы для групп захвата
<code>smatch.format(...)</code>	Формирует объекты <code>std::smatch</code> для вывода

В следующей ниже программе показан вывод из первых четырех групп захвата для разных регулярных выражений.

Группы захвата

```
// captureGroups.cpp
...

#include<regex>

...

using namespace std;

void showCaptureGroups(const string& regEx, const string& text){
    regex rgx(regEx);
    smatch smatch;
    if (regex_search(text, smatch, rgx)){
        cout << regEx << text << smatch[0] << " " << smatch[1]
            << " " << smatch[2] << " " << smatch[3] << endl;
    }
}

showCaptureGroups("abc+", "abccccc");
showCaptureGroups("(a+)(b+)", "aaabccc");
showCaptureGroups("((a+)(b+))", "aaabccc");
showCaptureGroups("(ab)(abc)+", "ababcabc");

...

reg Expr      text      smatch[0]  smatch[1]  smatch[2]  smatch[3]
abc+          abccccc   abccccc
(a+)(b+)(c+)  aaabccc   aaabccc   aaa        b          ccc
((a+)(b+)(c+)) aaabccc   aaabccc   aaabccc   aaa        b
(ab)(abc)+    ababcabc   ababcabc   ab         abc
```

std::sub_match

Группы захвата имеют тип `std::sub_match`. Как и для типа `std::match_results`, в языке C++ объявлены следующие ниже четыре псевдонима вспомогательно-го шаблонного класса `std::sub_match`:

```
typedef sub_match<const char*>          csub_match;
typedef sub_match<const wchar_t*>       wsub_match;
typedef sub_match<string::const_iterator> ssub_match;
typedef sub_match<wstring::const_iterator> wssub_match;
```

Группа захвата `cap` допускает проведение дальнейшего анализа.

Таблица. Интерфейс класса `std::sub_match` на примере объекта `cap`

Функция-член	Описание
<code>cap.matched()</code>	Указывает, было ли это совпадение успешным
<code>cap.first()</code> и <code>cap.end()</code>	Возвращает начальный и конечной итераторы последовательности символов
<code>cap.length()</code>	Возвращает длину группы захвата
<code>cap.str()</code>	Возвращает группу захвата в виде строкового объекта
<code>cap.compare(other)</code>	Сравнивает текущую группу захвата с другой группой захвата

Ниже приведен фрагмент исходного кода, показывающий взаимодействие между результатом поиска типа `std::match_results` и его группами захвата типа `std::sub_match`.

Взаимоигра между `std::match_results` и `std::sub_match`

```
// subMatch.cpp
...
#include <regex>

...

using std::cout;

std::string privateAddress="192.168.178.21";

std::string regEx(R"((\d{1,3})\.(\d{1,3})\.(\d{1,3})\.(\d{1,3}))");
std::regex rgx(regEx);
std::smatch smatch;

if (std::regex_match(privateAddress, smatch, rgx)){
    for (auto cap: smatch){
        cout << "capture group: " << cap << '\n';
    }
}
```

```

if (cap.matched){
    std::for_each(cap.first, cap.second, [](int v){
        cout << std::hex << v << " ";});
    cout << '\n';
    }
}
}

```

...

```

capture group: 192.168.178.21
31 39 32 2e 31 36 38 2e 31 37 38 2e 32 31

```

```

capture group: 192
31 39 32

```

```

capture group: 168
31 36 38

```

```

capture group: 178
31 37 38

```

```

capture group: 21
32 31

```

Регулярное выражение `regEx` расшифровывает IPv4-адрес (IP-адрес в 4-й версии). Оно извлекает компоненты адреса с помощью групп захвата. Наконец, группы захвата и ASCII-символы выводятся на консоль в шестнадцатеричном виде.

Совпадение

Шаблонная функция `std::regex_match` выявляет совпадение текста с текстовым шаблоном. Результат поиска `std::match_results`.

В приведенном ниже фрагменте исходного кода показаны три простых варианта применения шаблонной функции `std::regex_match`: с последовательностью символов C, последовательностью символов C++ и диапазоном, возвращая только булеву величину. Для объектов `std::match_results` соответствующим образом доступны все три варианта.

```
std::match
```

```

// match.cpp
...
#include <regex>

```

```

...

std::string numberRegEx(R"([-+]?([0-9]*\.[0-9]+|[0-9]+))");
std::regex rgx(numberRegEx);

const char* numChar{"2011"};
if (std::regex_match(numChar, rgx)){
    std::cout << numChar << "is a number." << '\n';
}
// 2011 is a number.

const std::string numStr{"3.14159265359"};
if (std::regex_match(numStr, rgx)){
    std::cout << numStr << " is a number." << '\n';
}
// 3.14159265359 is a number.

const std::vector<char> numVec{'-', '2', '.', '7', '1', '8', '2',
                               '8', '1', '8', '2', '8'};
if (std::regex_match(numVec.begin(), numVec.end(), rgx)){
    for (auto c: numVec){ std::cout << c ;};
    std::cout << "is a number." << '\n';
}
// -2.718281828 is a number

```

Поиск

Шаблонная функция `std::regex_search` выполняет проверку на наличие текстового шаблона в тексте. Данная функция может использоваться с объектом `std::match_results` и без него и применяться к последовательности символов C, последовательности символов C++ либо диапазону.

В приведенном ниже примере показано, как использовать функцию `std::regex_search` с текстом, имеющим типы `const char*`, `std::string`, `const wchar_t*` и `std::wstring`.

std::search

```

// search.cpp
...
#include <regex>
...

// держатель регулярного выражения для времени
std::regex crgx("([01]?[0-9]|2[0-3]):[0-5][0-9]");

// const char*

```

```

std::cmatch cmatch;

const char* ctime{"Now it is 23:10." };
if (std::regex_search(ctime, cmatch, crgx)){
    std::cout << ctime << '\n';           // Now it is 23:10.
    std::cout << "Time: " << cmatch[0] << '\n'; // Time: 23:10
}

// std::string
std::smatch smatch;

std::string stime{"Now it is 23:25." };
if (std::regex_search(stime, smatch, crgx)){
    std::cout << stime << '\n';           // Now it is 23:25.
    std::cout << "Time: " << smatch[0] << '\n'; // Time: 23:25
}

// держатель регулярного выражения для времени
std::wregex wrgx(L"([01]?[0-9]|2[0-3]):[0-5][0-9]");

// const wchar_t*
std::wcmatch wcmatch;

const wchar_t* wctime{L "Now it is 23:47." };
if (std::regex_search(wctime, wcmatch, wrgx)){
    std::wcout << wctime << '\n';           // Now it is 23:47.
    std::wcout << "Time: " << wcmatch[0] << '\n'; // Time: 23:47
}

// std::wstring
std::wsmatch wsmatch;

std::wstring wstime{L "Now it is 00:03." };
if (std::regex_search(wstime, wsmatch, wrgx)){
    std::wcout << wstime << '\n';           // Now it is 00:03.
    std::wcout << "Time: " << wsmatch[0] << '\n'; // Time: 00:03
}

```

Замена

Шаблонная функция `std::regex_replace` заменяет последовательности символов в тексте, совпадающие с текстовым шаблоном. При вызове `std::regex_replace(text, regex, replString)` она возвращает свой результат в простой форме в виде строкового объекта. Данная функция заменяет вхождение регулярного

выражения `regex` в тексте на подставляемое содержимое строкового объекта `replString`.

`std::replace`

```
// replace.cpp
...
#include <regex>

...

using namespace std;

string future{"Future"};
string unofficialName{"The unofficial name of the new C++ standard is C++0x."};

regex rgxCpp{R"(C\+\+0x)"};
string newCppName{"C++11"};
string newName{regex_replace(unofficialName, rgxCpp, newCppName)};

regex rgxOff{"unofficial"};
string makeOfficial{"official"};
string officialName{regex_replace(newName, rgxOff, makeOfficial)};

cout << officialName << endl;
// The official name of the new C++ standard is C++11
```

Дополнительно к простой версии шаблонной функции `std::regex_replace` в языке C++ есть версия, работающая на диапазонах. Она позволяет вставлять модифицированную последовательность символов непосредственно в другую последовательность символов:

```
typedef basic_regex<char> regex;
std::string str2;
std::regex_replace(std::back_inserter(str2),
                  text.begin(), text.end(), regex, replString);
```

Все варианты шаблонной функции `std::regex_replace` имеют дополнительный опциональный параметр. Если задать параметру значение `std::regex_constants::format_no_copy`, то будет получена часть текста, совпадающая с регулярным выражением. Несовпадающий текст не копируется. Если задать параметру значение `std::regex_constants::format_first_only`, то шаблонная функция `std::regex_replace` будет применена всего один раз.

Форматирование

Шаблонная функция `std::regex_replace` и функция-член `std::match_results::format` в сочетании с группами захвата позволяют форматировать текст. Для вставки значения можно использовать форматный литерал вместе с местозаполнителем.

Ниже приведены обе возможности:

Форматирование с использованием регулярного выражения

```
// format.cpp
...
#include <regex>

...

std::string future{"Future"};
const std::string unofficial{"unofficial, C++0x"};
const std::string official{"official, C++11"};

std::regex regValues{"(.*),(.*)"};
std::string standardText{"The $1 name of the new C++ standard is $2."};
std::string textNow= std::regex_replace(unofficial, regValues, standardText);
std::cout << textNow << '\n';
// The unofficial name of the new C++ standard is C++0x.

std::smatch smatch;
if (std::regex_match(official, smatch, regValues)){
    std::cout << smatch.str(); // official,C++11
    std::string textFuture= smatch.format(standardText);
    std::cout << textFuture << '\n';
}
// The official name of the new C++ standard is C++11.
```

При вызове функции `std::regex_replace(unofficial, regValues, standardText)` из строкового объекта `unofficial` извлекается текст, совпадающий с первой и второй группами захвата в регулярном выражении `regValues`. Затем местозаполнители `$1` и `$2` в тексте `standardText` заменяются извлеченными значениями. Стратегия функции `smatch.format(standardText)` аналогична, но есть одно отличие: при форматировании строкового объекта создание результатов поиска `smatch` отделено от их использования.

Помимо групп захвата, язык C++ поддерживает дополнительные форматные управляющие последовательности. Их можно применять в форматных литералах.

Таблица. Форматные управляющие последовательности

Форматная управляющая последовательность	Описание
\$&	Возвращает полное совпадение (нулевая группа захвата)
\$\$	Возвращает \$
\$` (обратный штрих)	Возвращает текст перед общим совпадением
\$^` (прямой штрих)	Возвращает текст после общего совпадения
'\$ i'	Возвращает i-ю группу захвата

Повторный поиск

С помощью итераторных шаблонных классов `std::regex_iterator` и `std::regex_token_iterator` довольно удобно перебирать совпадающие тексты. Итераторный класс `std::regex_iterator` поддерживает совпадения и их группы захватов, а итераторный класс `std::regex_token_iterator` поддерживает дополнительные функциональности. В частности, можно обращаться к компонентам каждого захвата. Использование отрицательного индекса позволяет обращаться к тексту между совпадениями.

`std::regex_iterator`

В языке C++ объявлены следующие ниже четыре псевдонима итераторного шаблонного класса `std::regex_iterator`.

```
typedef cregex_iterator    regex_iterator<const char*>
typedef wregex_iterator    regex_iterator<const wchar_t*>
typedef sregex_iterator    regex_iterator<std::string::const_iterator>
typedef wsregex_iterator    regex_iterator<std::wstring::const_iterator>
```

Итераторный шаблонный класс `std::regex_iterator` можно использовать для подсчета вхождений слов в текст:

```
std::regex_iterator
```

```
// regexIterator.cpp
```

```
...
```

```
#include <regex>
```

```
...
```

```
using std::cout;
```

```
std::string text{"That's a (to me) amazingly frequent question. It may be t\
he most frequently asked question. Surprisingly, C++11 feels like a ne\"
```

w language: The pieces just fit together better than they used to, and I find a higher-level style of programming more natural than before and as efficient as ever." };

```
std::regex wordReg{R"(\w+)"};
std::sregex_iterator wordItBegin(text.begin(), text.end(), wordReg);
const std::sregex_iterator wordItEnd;
std::unordered_map<std::string, std::size_t> allWords;
for (; wordItBegin != wordItEnd; ++wordItBegin){
    ++allWords[wordItBegin->str()];
}
for (auto wordIt: allWords) cout << "(" << wordIt.first << ":"
    << wordIt.second << ")";
// (as:2)(of:1)(level:1)(find:1)(ever:1)(and:2)(natural:1) ...
```

Слово состоит как минимум из одного символа (\w+). Это регулярное выражение используется для задания начального итератора wordItBegin и конечного итератора wordItEnd. Итерации по совпадениям происходят в цикле for. Каждое слово увеличивает счетчик: ++allWords[wordItBegin->str()]. Если слова еще нет в allWords, то это слово создается со счетчиком, равным 1.

std::regex_token_iterator

В языке C++ объявлены следующие ниже четыре псевдонима итераторного шаблонного класса std::regex_token_iterator:

```
typedef cregex_iterator      regex_iterator<const char*>
typedef wcregex_iterator    regex_iterator<const wchar_t*>
typedef sregex_iterator     regex_iterator<std::string::const_iterator>
typedef wsregex_iterator    regex_iterator<std::wstring::const_iterator>
```

Итераторный шаблонный класс std::regex_token_iterator позволяет использовать индексы, чтобы явно указывать интересующие группы захвата. Если индекс не указан, то будут получены все группы захвата. Кроме того, можно запрашивать конкретные группы захвата, используя соответствующий индекс. Индекс -1 является особым: его можно использовать для обращения к тексту между совпадениями.

std::regex_token_iterator

```
// tokenIterator.cpp
...
```

```
using namespace std;
```

```
std::string text{"Pete Becker, The C++ Standard Library Extensions, 2006:"
    "Nicolai Josuttis, The C++ Standard Library, 1999:"
```

```
"Andrei Alexandrescu, Modern C++ Design, 2001"};
```

```
regex regBook(R"((\w+)\s(\w+),([\w\s\+]*),(\d{4}))");
sregex_token_iterator bookItBegin(text.begin(), text.end(), regBook);

const sregex_token_iterator bookItEnd;
while (bookItBegin != bookItEnd){
    cout << *bookItBegin++ << endl;
}
// Pete Becker, The C++ Standard Library Extensions, 2006
// Nicolai Josuttis, The C++ Standard Library, 1999
sregex_token_iterator bookItNameIssueBegin(text.begin(), text.end(),
                                           regBook, {{2,4}});
const sregex_token_iterator bookItNameIssueEnd;
```

16. Потоки ввода и вывода



Циппи плывет на лодке по бурной реке

Краткий обзор

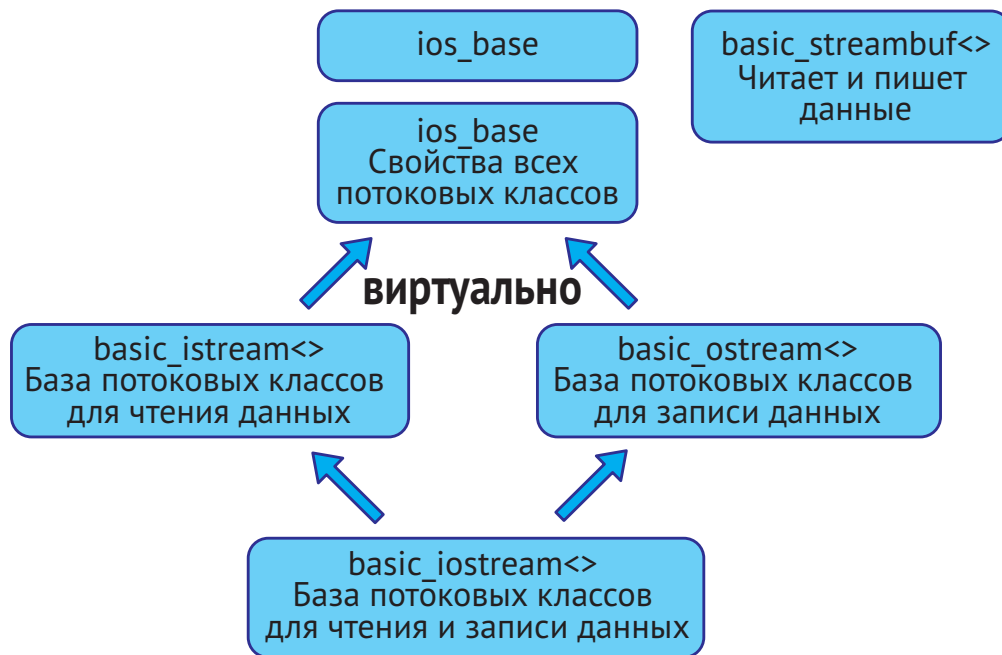
Потоки ввода и вывода данных¹ позволяют общаться с внешним миром. Поток данных представляет собой бесконечный поток символов, в который можно вталкивать данные или откуда их можно выталкивать. Вталкивание называется записью в поток, а выталкивание – чтением из потока.

Потоки ввода и вывода данных:

- использовались задолго до появления первого стандарта C++ (C++98) в 1998 году;
- предназначены для обеспечения расширяемости;
- реализованы в соответствии с парадигмами объектно ориентированного и обобщенного программирования.

¹ См. <http://en.cppreference.com/w/cpp/header/iostream>.

Иерархия



`basic_streambuf<>`

Читает и пишет данные.

`ios_base`

Свойства всех потоковых классов, не зависящие от типа символов.

`basic_ios<>`

Свойства всех потоковых классов, зависящие от типа символов.

`basic_istream<>`

База потоковых классов для чтения данных.

`basic_ostream<>`

База потоковых классов для записи данных.

`basic_iostream<>`

База потоковых классов для чтения и записи данных.

В иерархии классов есть псевдонимы символьных типов `char` и `wchar_t`. Имена, не начинающиеся с `w`, являются псевдонимами типа `char`; имена, начинающиеся с `w`, – псевдонимами типа `wchar_t`.

Базовые классы шаблонного класса `std::basic_iostream<>` виртуально производны от базового класса `std::basic_ios<>`, поэтому шаблонный класс `std::basic_iostream<>` имеет только один экземпляр базового класса `std::basic_ios`.

Функции ввода и вывода

Потоковые классы `std::istream` и `std::ostream` часто используются для чтения и записи данных. Использование классов `std::istream` предусматривает наличие заголовка `<istream>`; использование классов `std::ostream` нуждается в наличии заголовка `<ostream>`. Благодаря заголовку `<iostream>` можно использовать оба варианта. Класс `std::istream` объявлен как псевдоним вспомогательного шаблонного класса `basic_istream` и символьного типа `char`, соответственно, класс `std::ostream` объявлен как псевдоним вспомогательного шаблонного класса `basic_ostream` и символьного типа `char`:

```
typedef basic_istream<char>    istream;
typedef basic_ostream<char>    ostream;
```

Для удобства работы с клавиатурой и монитором в языке C++ предопределены четыре потоковых объекта.

Таблица. Четыре предопределенных потоковых объекта

Потоковый объект	Аналог в С	Устройство	Буферизация
<code>std::cin</code>	<code>stdin</code>	клавиатура	да
<code>std::cout</code>	<code>stdout</code>	консоль	да
<code>std::cerr</code>	<code>stderr</code>	консоль	нет
<code>std::clog</code>		консоль	да



Потоковые объекты также доступны для типа `wchar_t`

Пять потоковых объектов для типов `wchar_t`, `std::wcin`, `std::wcout`, `std::wcerr` и `std::wclog` далеко не так активно используются, как их аналоги для `char`, и по этой причине они рассматриваются в книге лишь в малой степени.

В языке C++ существует достаточно потоковых объектов, чтобы написать программу, которая читает из командной строки и возвращает сумму.

Потоковые объекты

```
// IOStreams.cpp
...

#include <iostream>

int main(){
    std::cout << "Type in your numbers";
    std::cout << "(Quit with an arbitrary character): " << std::endl;
    // 2000 <Enter> 11 <a>
    int sum{0};
```

```
int val;
while (std::cin >> val) sum += val;
std::cout << "Sum: " << sum;           // Sum: 2011
}
```

В приведенной выше небольшой программе используются оператор вставки данных в поток << и оператор извлечения данных из потока >>, а также манипулятор потока `std::endl`.

Оператор вставки данных в поток << вталкивает символы в поток вывода `std::cout`; оператор извлечения данных из потока >> выталкивает символы из потока ввода `std::cin`. Из операторов вставки и извлечения можно строить цепочки, так как оба оператора возвращают ссылку на себя. Манипулятор потока `std::endl` помещает символ '\n' в поток вывода `std::cout` и очищает буфер вывода.

Ниже приведены наиболее часто используемые манипуляторы потоком.

Таблица. Наиболее часто используемые манипуляторы потоком

Манипулятор	Тип потока	Описание
<code>std::endl</code>	вывод	Вставляет символ новой строки и очищает поток данных
<code>std::flush</code>	вывод	Немедленно сбрасывает поток данных
<code>std::ws</code>	ввод	Отбрасывает начальные пробельные символы

Извлечение данных (ввод)

В языке C++ можно читать из потока ввода данных двумя способами: форматированно с помощью извлекающего >> и неформатированно с помощью за-данных в явной форме функций-членов.

Форматированное извлечение

Оператор извлечения из потока >>:

- предопределен для всех встроенных типов и строковых объектов;
- может быть реализован для пользовательских типов данных;
- может корректироваться конкретизаторами формата.



Поток ввода данных `std::cin` по умолчанию игнорирует начальные пробельные символы

```
#include <iostream>

...
int a, b;
std::cout << "Two natural numbers: " << '\n';
std::cin >> a >> b;           // < 2000 11>
std::cout << "a: " << a << " b: " << b;
```

Неформатированное извлечение

Для неформатированного извлечения из потока ввода данных `is` имеется несколько функций-членов.

Таблица. Неформатированное извлечение из потока ввода данных

Функция-член	Описание
<code>is.get(ch)</code>	Читает один символ в аргумент <code>ch</code>
<code>is.get(buf, num)</code>	Читает не более <code>num</code> символов в буфер <code>buf</code>
<code>is.getline(buf, num[, delim])</code>	Читает не более <code>num</code> символов в буфер <code>buf</code> . Опционально использует разделитель строк <code>delim</code> (по умолчанию <code>\n</code>)
<code>is.gcount()</code>	Возвращает количество последних символов, извлеченных из потока ввода данных <code>is</code> неформатированной операцией
<code>is.ignore(streamsize sz=1, int delim=end-of-file)</code>	Игнорирует <code>sz</code> символов вплоть до разделителя <code>delim</code>
<code>is.peek()</code>	Получает один символ из потока ввода данных <code>is</code> без его потребления
<code>is.unget()</code>	Вталкивает последний прочитанный символ назад в поток ввода данных <code>is</code>
<code>is.putback(ch)</code>	Вталкивает символ <code>ch</code> в поток ввода данных <code>is</code>



Конкретный класс `std::string` имеет функцию-член `getline`

Функция-член `getline` конкретного класса `std::string` имеет большое преимущество перед функцией-членом `getline` шаблонного класса `istream`. Класс `std::string` заботится о своей памяти автоматически. Напротив, при использовании функции `is.getline(buf, num)` необходимо резервировать память под буфер `buf`.

```
// inputUnformatted.cpp
...
#include <iostream>

...

std::string line;
std::cout << "Write a line: " << '\n';

std::getline(std::cin, line);    // <Only for testing purpose.>
std::cout << line << '\n';      // Only for testing purpose.

std::cout << "Write numbers, separated by;" << '\n';
```

```
while (std::getline(std::cin, line, ';') ) {
    std::cout << line << " ";
}
// <2000;11;a>
// 2000 11
```

Вставка данных (вывод)

С помощью оператора вставки данных в поток << можно вталкивать символы в поток вывода данных.

Оператор вставки данных в поток <<:

- предопределен для всех встроенных типов и строковых объектов;
- может быть реализован для пользовательских типов данных;
- может корректироваться конкретизаторами формата.

Конкретизатор формата

Конкретизаторы формата позволяют в явной форме корректировать данные, получаемые на входе и генерируемые на выходе.



В качестве конкретизаторов формата широко используются манипуляторы

Конкретизаторы формата доступны в виде манипуляторов и флагов. В этой книге приводятся только манипуляторы, потому что их функциональность довольно схожа и манипуляторы более удобно использовать.

Манипуляторы как конкретизаторы формата

```
// formatSpecifier.cpp
...
#include <iostream>
...

int num{2011};

std::cout.setf(std::ios::hex, std::ios::basefield);
std::cout << num << '\n';           // 7db
std::cout.setf(std::ios::dec, std::ios::basefield);
std::cout << num << '\n';           // 2011

std::cout << std::hex << num << '\n'; // 7db
std::cout << std::dec << num << '\n'; // 2011
```

В следующих ниже таблицах представлены важные конкретизаторы формата. Форматирование, установленное с помощью конкретизаторов формата, сохраняется (или «залипает») для последующих операций вывода до тех пор, пока не будет изменено, за исключением ширины поля, которая сбрасывается после каждого применения.

Манипуляторы без аргументов нуждаются в заголовке `<iostream>`; манипуляторы с аргументами нуждаются в заголовке `<iomanip>`.

Таблица. Вывод булевых величин на консоль

Манипулятор	Тип потока	Описание
<code>std::boolalpha</code>	ввод и вывод	Выводит булеву величину в виде слова
<code>std::noboolalpha</code>	ввод и вывод	Выводит булеву величину в виде числа (по умолчанию)

Таблица. Установка ширины поля и символа заполнения

Манипулятор	Тип потока	Описание
<code>std::setw(val)</code>	ввод и вывод	Устанавливает ширину поля равной значению <code>val</code>
<code>std::setfill(c)</code>	ввод и вывод	Устанавливает символ заполнения равным символу <code>c</code> (по умолчанию: пробел)

Таблица. Выравнивание текста

Манипулятор	Тип потока	Описание
<code>std::left</code>	вывод	Выравнивает выводимые данные влево
<code>std::right</code>	вывод	Выравнивает выводимые данные вправо
<code>std::internal</code>	вывод	Выравнивает знаки чисел влево, значения – вправо

Таблица. Положительные знаки и верхний либо нижний буквенный регистр

Манипулятор	Тип потока	Описание
<code>std::showpos</code>	вывод	Выводит положительные знаки
<code>std::noshowpos</code>	вывод	Не выводит положительные знаки (по умолчанию)
<code>std::uppercase</code>	вывод	Использует символы верхнего регистра для чисел (по умолчанию)
<code>std::lowercase</code>	вывод	Использует символы нижнего регистра для чисел

Таблица. Вывод основания числа на консоль

Манипулятор	Тип потока	Описание
<code>std::oct</code>	ввод и вывод	Использует натуральные числа в восьмеричном формате
<code>std::dec</code>	ввод и вывод	Использует натуральные числа в десятичном формате (по умолчанию)

Манипулятор	Тип потока	Описание
<code>std::hex</code>	ввод и вывод	Использует натуральные числа в шестнадцатеричном формате
<code>std::showbase</code>	вывод	Выводит основание числа
<code>std::noshowbase</code>	вывод	Не выводит основание числа (по умолчанию)

Для чисел с плавающей точкой существуют особые правила:

- количество значащих цифр (цифр после точки) по умолчанию равно шести;
- если количество значащих цифр недостаточно велико, то число выводится в научной нотации;
- начальные и замыкающие нули не выводятся;
- если возможно, десятичная точка не выводится.

Таблица. Числа с плавающей точкой

Манипулятор	Тип потока	Описание
<code>std::setprecision(val)</code>	вывод	Устанавливает прецизионность вывода равным значению <code>val</code>
<code>std::showpoint</code>	вывод	Выводит десятичную точку
<code>std::noshowpoint</code>	вывод	Не выводит десятичную точку (по умолчанию)
<code>std::fixed</code>	вывод	Выводит число с плавающей точкой в десятичном формате
<code>std::scientific</code>	вывод	Выводит число с плавающей точкой в научном формате
<code>std::hexfloat</code>	вывод	Выводит число с плавающей точкой в шестнадцатеричном формате
<code>std::defaultfloat</code>	вывод	Выводит число с плавающей точкой в стандартной нотации

Конкретизатор формата

```
// formatSpecifierOutput.cpp
...
#include <iomanip>
#include <iostream>
...

std::cout.fill('#');
std::cout << -12345;
std::cout << std::setw(10) << -12345;           // #####-12345
std::cout << std::setw(10) << std::left << -12345; // -12345####
std::cout << std::setw(10) << std::right << -12345; // #####-12345
```

```

std::cout << std::setw(10) << std::internal << -12345; //-####12345

std::cout << std::oct << 2011;           // 3733
std::cout << std::hex << 2011;           // 7db

std::cout << std::showbase;
std::cout << std::dec << 2011;           // 2011
std::cout << std::oct << 2011;           // 03733
std::cout << std::hex << 2011;           // 0x7db

std::cout << 123.456789;                 // 123.457
std::cout << std::fixed;
std::cout << std::setprecision(3) << 123.456789; // 123.457
std::cout << std::setprecision(6) << 123.456789; // 123.456789
std::cout << std::setprecision(9) << 123.456789; // 123.456789000

std::cout << std::scientific;
std::cout << std::setprecision(3) << 123.456789; // 1.235e+02
std::cout << std::setprecision(6) << 123.456789; // 1.234568e+02
std::cout << std::setprecision(9) << 123.456789; // 1.234567890e+02

std::cout << std::hexfloat;
std::cout << std::setprecision(3) << 123.456789; // 0x1.edd3c07ee0b0bp+6
std::cout << std::setprecision(6) << 123.456789; // 0x1.edd3c07ee0b0bp+6
std::cout << std::setprecision(9) << 123.456789; // 0x1.edd3c07ee0b0bp+6

std::cout << std::defaultfloat;
std::cout << std::setprecision(3) << 123.456789; // 123
std::cout << std::setprecision(6) << 123.456789; // 123.457
std::cout << std::setprecision(9) << 123.456789; // 123.456789

```

Потоки данных

Поток данных представляет собой бесконечную последовательность байтов, в которую можно вталкивать и из которой можно выталкивать данные. Строковые потоки и файловые потоки обеспечивают возможность прямого взаимодействия строковых объектов и файлов с потоком данных.

Строковые потоки

Строковые потоки нуждаются в заголовке `<sstream>`. Они не связаны с потоком ввода либо вывода и хранят свои данные в строковом объекте.

В языке C++ имеются различные классы строковых потоков независимо от того, используется ли строковый поток для ввода либо вывода или с символьным типом `char` либо `wchar_t`:

`std::istringstream` и `std::wistringstream`

Строковый поток для входных данных типа `char` и `wchar_t`.

`std::ostringstream` и `std::wostringstream`

Строковый поток для выходных данных типа `char` и `wchar_t`.

`std::stringstream` и `std::wstringstream`

Строковый поток для входных либо выходных данных типа `char` и `wchar_t`.

Типичные операции на строковом потоке таковы:

- запись данных в строковый поток:

```
std::stringstream os;
os << "New String";
os.str("Another new String");
```
- чтение данных из строкового потока:

```
std::stringstream os;
std::string str;
os >> str;
str= os.str();
```
- очистка строкового потока:

```
std::stringstream os;
os.str("");
```

Строковые потоки часто используются для типобезопасной конвертации между строковыми литералами и числами:

Конвертация строкового потока

```
// stringStreams.cpp
...
#include <sstream>

...

template <typename T>
T StringTo ( const std::string& source ){
    std::istringstream iss(source);
    T ret;
    iss >> ret;
    return ret;
}

template <typename T>
```

```

std::string ToString(const T& n){
    std::ostringstream tmp ;
    tmp << n;
    return tmp.str();
}

std::cout << "5= " << ToString<int>("5");           // 5
std::cout << "5 + 6= " << ToString<int>("5") + 6;    // 11
std::cout << ToString(ToString<int>("5") + 6 );      // "11"
std::cout << "5e10: " << std::fixed << ToString<double>("5e10"); // 50000000000

```

Файловые потоки

Файловые потоки позволяют работать с файлами. Они автоматически управляют временем жизни своего файла. Для них необходим заголовок `<fstream>`.

В языке C++ имеются различные классы файловых потоков независимо от того, используется файловый поток для ввода либо вывода или с символьным типом `char` либо `wchar_t`:

`std::ifstream` и `std::wifstream`

Файловый поток для входных данных типа `char` и `wchar_t`.

`std::ofstream` и `std::wofstream`

Файловый поток для выходных данных типа `char` и `wchar_t`.

`std::fstream` и `std::wfstream`

Файловый поток для входных и выходных данных типа `char` и `wchar_t`.

`std::filebuf` и `std::wfilebuf`

Буфер данных типа `char` и `wchar_t`.



Установка указателя позиции в файле

Файловые потоки, которые используются для чтения и записи, должны устанавливать указатель позиции в файле после изменения контекстов.

Флаги позволяют задавать режим открытия файлового потока.

Таблица. Флаги режимов открытия файлового потока

Флаг	Описание
<code>std::ios::in</code>	Открывает файловый поток для чтения (по умолчанию для <code>std::ifstream</code> и <code>std::wifstream</code>)

Флаг	Описание
<code>std::ios::out</code>	Открывает файловый поток для записи (по умолчанию для <code>std::ofstream</code> и <code>std::wofstream</code>)
<code>std::ios::app</code>	Добавляет символ в конец файлового потока
<code>std::ios::ate</code>	Устанавливает начальную позицию указателя позиции в файле в конце файлового потока
<code>std::ios::trunc</code>	Удаляет изначальный файл
<code>std::ios::binary</code>	Подавляет интерпретацию управляющей последовательности в файловом потоке

Копирование файла с именем `in` в файл с именем `out` с помощью файлового буфера `in.rdbuf()` выполняется довольно просто. В приведенный ниже короткий пример необходимо включить исправление ошибок.

```
#include <fstream>
...
std::ifstream in("inFile.txt");
std::ofstream out("outFile.txt");
out << in.rdbuf();
```

В следующей ниже таблице приведено сравнение режимов открытия файла в языке C++ с такими же режимами в языке C.

Таблица. Открытие файла в языке C++ и языке C

Режим C++	Описание	Режим C
<code>std::ios::in</code>	Читает файл	"r"
<code>std::ios::out</code>	Пишет файл	"w"
<code>std::ios::out std::ios::app</code>	Добавляет в конец файла	"a"
<code>std::ios::in std::ios::out</code>	Читает и пишет файл	"r+"
<code>std::ios::in std::ios::out std::ios::trunc</code>	Пишет и читает файл	"w+"

Файл должен существовать с режимами "r" и "r+". И наоборот, файл создается с режимами "a" и "w+". Файл перезаписывается в режиме "w".

Управление временем жизни файлового потока можно осуществлять явным образом.

Таблица. Управление временем жизни файлового потока

Флаг	Описание
<code>infile.open(name)</code>	Открывает файл <code>name</code> для чтения
<code>infile.open(name, flags)</code>	Открывает файл <code>name</code> с заданными флагами для чтения
<code>infile.close()</code>	Закрывает файл <code>name</code>
<code>infile.is_open()</code>	Проверяет, что файл открыт

Произвольный доступ

Произвольный доступ позволяет устанавливать указатель позиции в файле в любом месте.

При создании файлового потока указатель позиции в файле указывает на начало файла. Позицию можно корректировать с помощью функций-членов объекта – файлового потока `file`.

Таблица. Навигация в файловом потоке

Функция-член	Описание
<code>file.tellg()</code>	Возвращает позицию чтения в файле <code>file</code>
<code>file.tellp()</code>	Возвращает позицию записи в файле <code>file</code>
<code>file.seekg(pos)</code>	Устанавливает позицию чтения в файле <code>file</code> равной позиции <code>pos</code>
<code>file.seekp(pos)</code>	Устанавливает позицию записи в файле <code>file</code> равной позиции <code>pos</code>
<code>file.seekg(off, gpos)</code>	Устанавливает позицию чтения в файле <code>file</code> равной смещению <code>off</code> относительно позиции <code>gpos</code>
<code>file.seekp(off, gpos)</code>	Устанавливает позицию записи в файле <code>file</code> равной смещению <code>off</code> относительно позиции <code>gpos</code>

Смещение `off` должно быть числом. Позиция `gpos` может иметь три значения:

`std::ios::beg`

Позиция в начале файла.

`std::ios::cur`

Позиция в текущем месте.

`std::ios::end`

Позиция в конце файла.



Соблюдение границ файла

При произвольном обращении к файлу среда исполнения C++ не проверяет границы файла. Поведение при чтении или записи данных вне границ будет неопределенным.

Произвольный доступ

```
// randomAccess.cpp
...
#include <fstream>
...

void writeFile(const std::string name){
```

```
std::ofstream outFile(name);
if (!outFile){
    std::cerr << "Could not open file " << name << '\n';
    exit(1);
}
for (unsigned int i= 0; i < 10 ; ++i){
    outFile << i << " 0123456789" << '\n';
}

std::string random{"random.txt"};
writeFile(random);
std::ifstream inFile(random);

if (!inFile){
    std::cerr << "Could not open file " << random << '\n';
    exit(1);
}

std::string line;

std::cout << inFile.rdbuf();
// 0      0123456789
// 1      0123456789
...
// 9      0123456789

std::cout << inFile.tellg() << '\n'; // 200

inFile.seekg(0); // inFile.seekg(0, std::ios::beg);
getline(inFile, line);
std::cout << line; // 0 0123456789

inFile.seekg(20, std::ios::cur);
getline(inFile, line);
std::cout << line; // 2 0123456789

inFile.seekg(-20, std::ios::end);
getline(inFile, line);
std::cout << line; // 9 0123456789
```

Состояние потока данных

Флаги представляют состояние объекта-потока данных `stream`. Функции-члены для работы с этими флагами нуждаются в заголовке `<iostream>`.

Таблица. Состояние потока

Флаг	Запрос флага	Описание
<code>std::ios::goodbit</code>	<code>stream.good()</code>	Бит не установлен
<code>std::ios::eofbit</code>	<code>stream.eof()</code>	Установлен бит конца файла
<code>std::ios::failbit</code>	<code>stream.fail()</code>	Ошибка
<code>std::ios::badbit</code>	<code>stream.bad()</code>	Неопределенное поведение

Ниже приведены примеры условий, вызывающих разные состояния потока данных:

`std::ios::eofbit`

- чтение после последнего допустимого символа;

`std::ios::failbit`

- ложно отформатированное чтение;
- чтение за пределами последнего допустимого символа;
- открытие файла прошло с ошибкой;

`std::ios::badbit`

- размер буфера потока данных невозможно скорректировать;
- кодовая конвертация буфера потока данных прошла неправильно;
- часть потока данных сгенерировала исключение.

Функция-член `stream.fail()` возвращает `true`, если установлен флаг состояния `std::ios::failbit` либо `std::ios::badbit`, либо оба флага.

Состояние потока можно читать и устанавливать.

`stream.clear()`

Данная функция-член инициализирует флаги и переводит поток данных в состояние `goodbit`.

`stream.clear(sta)`

Данная функция-член инициализирует флаги и переводит поток данных в состояние `sta`.

`stream.rdstate()`

Данная функция-член возвращает текущее состояние.

`stream.setstate(fla)`

Данная функция-член устанавливает дополнительный флаг `fla`.

Операции на потоке данных работают только в том случае, если поток находится в состоянии `goodbit`. Если поток находится в состоянии `badbit`, то вернуть его в состояние `goodbit` невозможно.

Состояние потока данных

```
// streamState.cpp
...
#include <iostream>

...

std::cout << std::cin.fail() << '\n';    // false

int myInt;

while (std::cin >> myInt){ // <a>
    std::cout << myInt << '\n'; //
    std::cout << std::cin.fail() << '\n'; //
}

std::cin.clear();
std::cout << std::cin.fail() << '\n';    // false
```

Ввод символа `a` приводит к тому, что поток данных `std::cin` переходит в состояние `std::ios::failbit`, и поэтому вывести на консоль символы `a` и `std::cin.fail()` невозможно. Сначала необходимо убедиться, что поток данных `std::cin` инициализирован.

Пользовательские типы данных

При выполнении перегрузки операторов ввода и вывода пользовательский тип данных ведет себя как встроенный тип данных.

```
friend std::istream& operator>> (std::istream& in, Fraction& frac);
friend std::ostream& operator<< (std::ostream& out, const Fraction& frac);
```

С целью правильного выполнения перегрузки операторов ввода и вывода необходимо помнить о нескольких правилах:

- для поддержания возможности выстраивать операции ввода и вывода в цепочку необходимо получать и возвращать потоки ввода и вывода данных по неконстантной ссылке;
- для получения доступа к приватным членам класса операторы ввода и вывода должны быть друзьями пользовательского типа данных;
- оператор извлечения из потока `>>` принимает тип данных в виде неконстантной ссылки;
- оператор вставки в поток `<<` принимает тип данных в виде константной ссылки.

Перегрузка операторов извлечения и вставки данных

```
// overloadingInOutput.cpp

class Fraction{
public:
    Fraction(int num= 0, int denom= 0):numerator(num), denominator(denom){}
    friend std::istream& operator>> (std::istream& in, Fraction& frac);
    friend std::ostream& operator<< (std::ostream& out, const Fraction& frac);
private:
    int numerator;
    int denominator;
};

std::istream& operator>> (std::istream& in, Fraction& frac){
    in >> frac.numerator;
    in >> frac.denominator;
    return in;
}

std::ostream& operator<< (std::ostream& out, const Fraction& frac){
    out << frac.numerator << "/" << frac.denominator;
    return out;
}

Fraction frac(3, 4);
std::cout << frac;           // 3/

std::cout << "Enter two numbers: ";
Fraction fracDef;

std::cin >> fracDef;          // <1 2>
std::cout << fracDef;         // 1/2
```

17. Форматирование



Циппи формирует кружку из глины

Краткий обзор

Библиотека для работы с форматированием¹ в стандарте C++20 представляет собой безопасную и расширяемую альтернативу семейству функций `printf` и расширяет потоки ввода и вывода данных. Данная библиотека нуждается в наличии заголовка `<format>`, а синтаксис форматного литерала соответствует синтаксису языка Python.

Благодаря библиотеке `std::format` формат можно конкретизировать с помощью форматного литерала, чтобы:

- выравнивать текст и назначать символ заполнения;
- назначать знак, ширину и прецизионность чисел;
- назначать тип данных.

Форматирующие функции

Язык C++ поддерживает различные форматирующие функции. К ним относятся элементарные форматирующие функции времени компиляции `std::format`, `std::format_to` и `std::format_to_n` и форматирующие функции времени исполнения `std::vformat` и `std::vformat_to` в сочетании со вспомогательной функцией `std::make_format_args`. Кроме того, еще имеются вспомогательные функции `std::print` и `std::println`.

В библиотеке `format` есть три элементарные форматирующие функции:

`std::format`

Данная функция возвращает отформатированный строковый литерал.

¹ См. <https://en.cppreference.com/w/cpp/utility/format>.

`std::format_to`

Данная функция пишет отформатированный строковый литерал посредством итератора вывода.

`std::format_to_n`

Данная функция пишет отформатированный строковый литерал посредством итератора вывода, но не более чем *n* символов.

```
#include <format>
```

```
....
```

```
std::format("{1} {0}!", "world", "Hello"); // Hello world!
```

```
std::string buffer;
```

```
std::format_to(std::back_inserter(buffer),  
    "Hello, C++{ }!\n", "20"); // Hello, C++20!
```

`std::vformat`, `std::vformat_to` и `std::make_format_args`

Для создания отформатированного строкового литерала в трех формирующих функциях `std::format`, `std::format_to` и `std::format_to_n` используется форматный литерал. Форматный литерал должен быть значением, вычисляемым во время компиляции. Соответственно, недопустимый форматный литерал приводит к ошибке, возникающей в фазе компиляции.

Для форматных литералов, применяемых во время исполнения, существуют альтернативные функции `std::vformat` и `std::vformat_to`, которые необходимо использовать в сочетании со вспомогательной функцией `std::make_format_args`.

```
#include <format>
```

```
...
```

```
std::string formatString = "{1} {0}!";  
std::vformat(formatString, std::make_format_args("world", "Hello")); // Hello world
```

`std::print` и `std::println`

Вспомогательные `std::print` и `std::println` пишут данные на консоль вывода. Функция `std::println` добавляет в вывод символ новой строки. Вдобавок обе функции позволяют писать в файловый поток вывода данных и поддерживают Юникод¹. При этом требуется вставка заголовка `<print>`.

```
#include <print>
```

```
...
```

```
std::print("{1} {0}!", "world", "Hello"); // prints "Hello world!"
```

```
std::ofstream outFile("testfile.txt");
```

¹ См. <https://en.wikipedia.org/wiki/Unicode>.

```
std::print(outFile, "{1} {0}!", "world", "Hello");
// writes "Hello world!" into the outFile
```



В остальной части этой главы используется функция `std::format`

В формирующих функциях времени компиляции `std::format`, `std::format_to` и `std::format_to_n`, формирующих функциях времени исполнения `std::vformat` и `std::vformat_n`, а также вспомогательных функциях `std::print` и `std::println` применяется одинаковый синтаксис форматного литерала. Ради простоты в остальной части этой главы используется функция `std::format`.

Синтаксис

Функция `std::format` имеет синтаксис: `std::format(FormatString, Args)`.

Форматный литерал `FormatString` состоит из:

- обыкновенных символов (кроме { и });
- управляющих последовательностей {{ и }}, которые заменяются на { и };
- полей замены.

Поле замены имеет формат:

- открывающий символ {:
 - идентификатор аргумента (опционально);
 - двоеточие :, за которым следует конкретизация формата (опционально);
- закрывающий символ }.

Идентификатор аргумента позволяет указывать индекс аргументов в `Args`. Идентификаторы начинаются с 0. Если ИД аргумента не указывается, то аргументы используются в том порядке, в котором они заданы. ИД аргумента должен использоваться либо во всех полях замены, либо ни в одном.

Шаблонный класс `std::formatter` и его конкретные подклассы конкретизируют формат для аргументов:

- базовые типы и строковые типы: стандартная конкретизация формата¹, основанная на конкретизации, принятой в языке Python²;
- хронотипы: хронологическая конкретизация формата³;
- другие типы: пользовательская конкретизация формата.

¹ См. https://en.cppreference.com/w/cpp/utility/format/formatter#Standard_format_specification.

² См. <https://docs.python.org/3/library/string.html#formatspec>.

³ См. https://en.cppreference.com/w/cpp/chrono/system_clock/formatter#Format_specification.

Конкретизация формата

В конкретизации формата можно указывать символ заполнения и выравнивание текста, знак, ширину, прецизионность (глубину точности) чисел и тип данных.

Символы заполнения и выравнивание текста

- Символ заполнения: по умолчанию пробел.
- Выравнивание:
 - <: слева;
 - >: справа;
 - ^: по центру.

```
char c = 120;
```

```
std::format("{:7}", 42);    // " 42"
std::format("{:7}", 'x');   // "x "
std::format("{:*<7}", 'x'); // "x*****"
std::format("{:*>7}", 'x'); // "*****x"
std::format("{:*^7}", 'x'); // "***x***"
std::format("{:7d}", c);    // " 120"
std::format("{:7}", true);  // "true "
```

Знак, ширина и прецизионность чисел

- Знак:
 - +: число получает знак;
 - -: отрицательные числа получают знак (по умолчанию);
 - пробел: положительные числа получают пробел.

```
double inf = std::numeric_limits<double>::infinity();
double nan = std::numeric_limits<double>::quiet_NaN();
```

```
std::format("{0:},{0:+},{0:-},{0: }", 1); // "1,+1,1, 1"
std::format("{0:},{0:+},{0:-},{0: }", -1); // "-1,-1,-1,-1"
std::format("{0:},{0:+},{0:-},{0: }", inf); // "inf,+inf,inf, inf"
std::format("{0:},{0:+},{0:-},{0: }", nan); // "nan,+nan,nan, nan"
```

- #:
 - использует альтернативный формат;
 - интегралы:
 - * двоичные числа: 0b;
 - * восьмеричные числа: 0;
 - * шестнадцатеричные числа: 0x;

- 0: заполняет начальными нулями:


```
std::format("{:+06d}", 120); // "+00120"
std::format("{:#0f}", 120)); // "120.000000"
std::format("{:0>15f}", 120)); // "00000120.000000"
std::format("{:#06x}", 0xa); // "0x000a"
```
- ширина: задает минимальную ширину;
- прецизионность: может применяться к числам с плавающей точкой и строковым литералам:
 - число с плавающей точкой: место после десятичной точки;
 - строковые литералы: максимальное количество символов.

```
double d = 20.11;
```

```
std::format!("{}", d); // " 20.11 "
std::format("{:10}", d); // " 20.11 "
std::format("{:10.3}", d); // " 20.1 "
std::format("{:.3}", "Only a test."); // " Onl "
```

Типы данных

Значения копируются в вывод, если не указано иное. Представление значения можно конкретизировать в явной форме:

- строковое представление: s;
- целочисленное представление:
 - b, B: двоичное;
 - d: десятичное;
 - o: восьмеричное;
 - x, X: шестнадцатеричное;
- символьное представление:
 - b, B, d, o, x, X: целое число;
- булево представление:
 - s: истина или ложь;
 - b, B, d, o, x, X: целое число;
- представление чисел с плавающей точкой:
 - e, E: научное;
 - f, F: десятичное.

Пользовательский форматизатор

Для того чтобы отформатировать пользовательский тип, необходимо конкретизировать шаблонный класс `std::formatter`¹ для своего пользовательского типа. Это, в частности, означает необходимость реализации функций-членов `parse` и `format`.

¹ См. <https://en.cppreference.com/w/cpp/utility/format/formatter>.

- `parse`:
 - принимает контекст структурного разбора;
 - разбирает контекст структурного разбора;
 - возвращает итератор до самого конца конкретизации формата;
 - в случае ошибки генерирует сообщение `std::format_error`.
- `format`:
 - получает форматируемое значение `t` и контекст формата `fc`;
 - форматирует `t` в соответствии с контекстом формата;
 - записывает вывод в `fc.out()`;
 - возвращает итератор, представляющий конец вывода.

Давайте применим теорию на практике и отформатируем контейнер `std::vector`.

Форматирование контейнера `std::vector`

Конкретизация шаблонного класса `std::formatter` выполняется максимально просто путем конкретизации формата, который применяется к каждому элементу контейнера.

```
// formatVector.cpp
#include <format>

...

template <typename T>
struct std::formatter<std::vector<T>> {

    std::string formatString;

    auto constexpr parse(format_parse_context& ctx) { // (3)
        formatString = "{: ";
        std::string parseContext(std::begin(ctx), std::end(ctx));
        formatString += parseContext;
        return std::end(ctx) - 1;
    }

    template <typename FormatContext>
    auto format(const std::vector<T>& v, FormatContext& ctx) {
        auto out= ctx.out();
        std::format_to(out, "[");
        if (v.size() > 0) std::format_to(out, formatString, v[0]);
        for (int i= 1; i < v.size(); ++i) {
            std::format_to(out, ", " + formatString, v[i]); // (1)
        }
        std::format_to(out, "]" ); // (2)
    }
};
```

```

    return std::format_to(out, "\n" );
}

};

...

std::vector<int> myInts{1, 2, 3, 4, 5};
std::cout << std::format("{:}", myInts);           // [1, 2, 3, 4, 5]      // (4)
std::cout << std::format("{:+}", myInts);           // [+1, +2, +3, +4, +5]
std::cout << std::format("{:03d}", myInts);         // [001, 002, 003, 004, 005]
std::cout << std::format("{:b}", myInts);           // [1, 10, 11, 100, 101] // (5)

std::vector<std::string> myStrings{"Only", "for", "testing"};
std::cout << std::format("{:}", myStrings);         // [Only, for, testing]
std::cout << std::format("{:.3}", myStrings);       // [Onl, for, tes]

```

Конкретизация для контейнера `std::vector` имеет функции-члены `parse` и `format`. Функция-член `parse`, по сути дела, создает форматный строковый объект `formatString`, применяемый к каждому элементу векторного контейнера `std::vector` (1 и 2). Контекст структурного разбора `ctx` (3) содержит символы между двоеточием (:) и закрывающими фигурными скобками (}). Указанная функция возвращает итератор до закрывающих фигурных скобок (}). Работа функции-члена `format` вызывает больший интерес. Контекст формата возвращает итератор вывода. Благодаря итератору вывода и функции `std::format_to`¹ элементы векторного контейнера `std::vector` разборчиво выводятся на консоль.

Элементы контейнера `std::vector` форматируются несколькими способами. Первая строка (4) выводит на консоль числа. Следующая строка перед каждым числом ставит знак, числа выравниваются по 3 символам, а 0 используется в качестве символа заполнения. Строка (5) выводит их в двоичном формате. Оставшиеся две строки выводят все строковые литералы вектора `std::vector`. Последняя строка усекает каждый строковый литерал до трех символов.

¹ См. https://en.cppreference.com/w/cpp/utility/format/format_to.

18. Файловая система



Циппи сортирует принадлежности для рисования

Краткий обзор

Библиотека `std::filesystem` для работы с файловой системой¹ была введена в стандарте C++17 и основана на библиотеке `boost::filesystem`². Некоторые из ее компонентов являются опциональными, в силу чего не вся функциональность библиотеки `std::filesystem` доступна в каждой реализации файловой системы. Например, FAT-32 не поддерживает символические ссылки.

В основе библиотеки лежат три понятия: файл, имя файла и путь.

- Файл – это объект, содержащий данные, в который можно писать или из которого можно читать. У файла есть имя и тип. Тип файла может быть каталогом, жесткой ссылкой, символической ссылкой или обычным файлом.
 - Каталог – это контейнер для хранения других файлов. Текущий каталог обозначается точкой «.»; две точки обозначают родительский каталог «..».
 - Жесткая ссылка связывает имя с существующим файлом.
 - Символическая ссылка связывает имя с путем, который может существовать.
 - Обычный файл – это запись в каталоге, которая не является ни каталогом, ни жесткой ссылкой, ни символической ссылкой.

¹ См. <http://en.cppreference.com/w/cpp/filesystem>.

² См. http://www.boost.org/doc/libs/1_65_1/libs/filesystem/doc/index.htm.

- Имя файла – это строковый литерал, представляющий файл, и то, какие символы разрешены, какой длины может быть имя и чувствительно ли имя к регистру символов, зависит от реализации.
- Путь – это последовательность записей, определяющая местоположение файла. Он имеет опциональное корневое имя, такое как «с:» в Windows, за которым следует корневой каталог, такой как «/» в Unix. Дополнительными частями могут быть каталоги, жесткие ссылки, символические ссылки или обычные файлы. Путь может быть абсолютным, каноническим либо относительным.
 - Абсолютный путь – это путь, идентифицирующий файл.
 - Канонический путь – это путь, не содержащий ни символических ссылок, ни относительных путей «.» (текущий каталог) или «..» (родительский каталог).
 - Относительный путь конкретизирует путь относительно места в файловой системе. Такие пути, как «.» (текущий каталог), «..» (родительский каталог) или «home/gainer», являются относительными путями. В Unix они не начинаются с корневого каталога «/».

Ниже приведен вводный пример работы с библиотекой `std::filesystem`.

Краткий обзор библиотеки `std::filesystem`

```
// filesystem.cpp
...
#include <filesystem>

...

namespace fs = std::filesystem;

std::cout << "Current path: " << fs::current_path() << '\n'; // (1)

std::string dir= "sandbox/a/b";
fs::create_directories(dir); // (2)

std::ofstream("sandbox/file1.txt");
fs::path symPath= fs::current_path() /= "sandbox"; // (3)
symPath /= "syml";
fs::create_symlink("a", "syml"); // (4)

std::cout << "fs::is_directory(dir): " << fs::is_directory(dir) << '\n';
std::cout << "fs::exists(symPath): " << fs::exists(symPath) << '\n';
std::cout << "fs::symlink(symPath): " << fs::is_symlink(symPath) << '\n';

for(auto& p: fs::recursive_directory_iterator("sandbox")){ // (5)
    std::cout << p << '\n';
}
fs::remove_all("sandbox");
```

Функция-член `fs::current_path()` (1) возвращает текущий путь. С помощью вспомогательной функции `std::filesystem::create_directories` можно создать иерархию каталогов (2). Оператор `/=` перегружен для пути (3), поэтому можно создать символическую ссылку непосредственно (4) либо проверить свойства файла. Вызов итераторной функции `recursive_directory_iterator` (5) позволяет выполнить рекурсивный обход каталогов.

Результат:

```
Current path: "/tmp/1469540273.75652"

fs::is_directory(dir): true
fs::exists(symPath): true
fs::symlink(symPath): true

"sandbox/syma"
"sandbox/file1.txt"
"sandbox/a"
"sandbox/a/b"
"sandbox/a/b/c"
```

Классы

Во многие классы библиотеки `std::filesystem` встроен тот или иной аспект файловой системы.

Таблица. Различные классы библиотеки `std::filesystem`

Класс	Описание
<code>path</code>	Представляет путь
<code>filesystem_error</code>	Определяет объект-исключение
<code>directory_entry</code>	Представляет запись в каталоге
<code>directory_iterator</code>	Определяет итератор по каталогам
<code>recursive_directory_iterator</code>	Определяет рекурсивный итератор каталогов
<code>file_status</code>	Хранит информацию о файле
<code>space_info</code>	Представляет информацию о файловой системе
<code>file_type</code>	Указывает тип файла
<code>perms</code>	Представляет разрешения на доступ к файлу
<code>perm_options</code>	Представляет опции для функции <code>permissions</code>
<code>copy_options</code>	Представляет опции для функций <code>copy</code> и <code>copy_file</code>
<code>directory_options</code>	Представляет опции для итераторных функций <code>directory_iterator</code> и <code>recursive_directory_iterator</code>
<code>file_time_type</code>	Представляет время файла

Манипулирование разрешениями на доступ к файлу

Разрешения на доступ к файлу представлены перечислением `std::filesystem::perms` в форме битовой маски `BitmaskType`¹, и, следовательно, им можно манипулировать с помощью побитовых операций. Разрешения на доступ основаны на переносимом интерфейсе операционных систем, POSIX².

Программа с веб-сайта en.cppreference.com³ показывает, как можно читать и манипулировать битами владельца, группы и другими (словарными) битами файла.

Разрешения на доступ к файлу

```
// perms.cpp
...
#include <filesystem>

...

namespace fs = std::filesystem;

void printPerms(fs::perms perm){
    std::cout << ((perm & fs::perms::owner_read) != fs::perms::none ? "r" : "-")
        << ((perm & fs::perms::owner_write) != fs::perms::none ? "w" : "-")
        << ((perm & fs::perms::owner_exec) != fs::perms::none ? "x" : "-")
        << ((perm & fs::perms::group_read) != fs::perms::none ? "r" : "-")
        << ((perm & fs::perms::group_write) != fs::perms::none ? "w" : "-")
        << ((perm & fs::perms::group_exec) != fs::perms::none ? "x" : "-")
        << ((perm & fs::perms::others_read) != fs::perms::none ? "r" : "-")
        << ((perm & fs::perms::others_write) != fs::perms::none ? "w" : "-")
        << ((perm & fs::perms::others_exec) != fs::perms::none ? "x" : "-")
        << '\n';
}

...

std::ofstream("rainer.txt");

std::cout << "Initial file permissions for a file: ";
printPerms(fs::status("rainer.txt").permissions());    // (1)

fs::permissions("rainer.txt", fs::perms::add_perms |    // (2)
    fs::perms::owner_all | fs::perms::group_all);
std::cout << "Adding all bits to owner and group: ";
printPerms(fs::status("rainer.txt").permissions());
```

¹ См. https://en.cppreference.com/w/cpp/named_req/BitmaskType.

² См. <https://en.wikipedia.org/wiki/POSIX>.

³ См. <https://en.cppreference.com/w/cpp/filesystem/perms>.

```
fs::permissions("rainer.txt", fs::perms::remove_perms | // (3)
  fs::perms::owner_write | fs::perms::group_write | fs::perms::others_write);
std::cout << "Removing the write bits for all: ";
printPerms(fs::status("rainer.txt").permissions());
```

Благодаря вызову функции `fs::status("rainer.txt").permissions()` можно получить разрешения для файла `rainer.txt` и вывести их на консоль в функции `printPerms` (1). После установки разрешения на добавление `std::filesystem::add_perms` можно добавить разрешения владельцу и группе файла (2). В качестве альтернативы можно установить константу `std::filesystem::remove_perms`, чтобы удалить разрешения (3).

```
Initial file permissions for a file: rw-r--r--
Adding all bits to owner and group:  rwxrwxr--
Removing the write bits for all:    r-xr-xr--
```

Функции-члены

Для работы с файловой системой существует множество функций, не являющихся членами классов.

Таблица. Функции-члены для работы с файловой системой

Функция-член	Описание
<code>absolute</code>	Составляет абсолютный путь
<code>canonical</code> и <code>weakly_canonical</code>	Составляет канонический путь
<code>relative</code> и <code>proximate</code>	Составляет относительный путь
<code>copy</code>	Копирует файлы или каталоги
<code>copy_file</code>	Копирует содержимое файла
<code>copy_symlink</code>	Копирует символическую ссылку
<code>create_directory</code> и <code>create_directories</code>	Создает новый каталог
<code>create_hard_link</code>	Создает жесткую ссылку
<code>create_symlink</code> и <code>create_directory_symlink</code>	Создает символическую ссылку
<code>current_path</code>	Возвращает текущий рабочий каталог
<code>exists</code>	Проверяет, не ссылается ли путь на существующий файл
<code>equivalent</code>	Проверяет, не ссылаются ли два пути на один и тот же файл
<code>file_size</code>	Возвращает размер файла
<code>hard_link_count</code>	Возвращает количество жестких ссылок на файл
<code>last_write_time</code>	Получает и устанавливает время последней модификации файла

Функция-член	Описание
<code>permissions</code>	Изменяет разрешения на доступ к файлу
<code>read_symlink</code>	Получает целевой файл или каталог, на который указывает символическая ссылка
<code>remove</code>	Удаляет файл или пустой каталог
<code>remove_all</code>	Рекурсивно удаляет файл или каталог со всем его содержимым
<code>rename</code>	Перемещает или переименовывает файл или каталог
<code>resize_file</code>	Изменяет размер файла путем его усеечения
<code>space</code>	Возвращает свободное пространство в файловой системе
<code>status</code>	Определяет атрибуты файла
<code>symlink_status</code>	Определяет атрибуты файла и проверяет целевой файл или каталог, на который указывает символическая ссылка
<code>temp_directory_path</code>	Возвращает каталог временных файлов

Чтение и установка времени последней записи файла

Благодаря вспомогательной функции `std::filesystem::last_write_time` можно читать и устанавливать время последней записи файла. Ниже приведен пример, основанный на примере `last_write_time`, взятом с веб-сайта `en.cppreference.com`¹.

Время записи файла

```
// fileTime.cpp
...
#include <filesystem>

...

namespace fs = std::filesystem;
using namespace std::chrono_literals;

...

fs::path path = fs::current_path() / "rainer.txt";
std::ofstream(path.c_str());
const auto fTime = fs::last_write_time(path);           // (1)
const auto sTime = std::chrono::clock_cast<std::chrono::system_clock>(fTime); // (2)
```

¹ См. http://en.cppreference.com/w/cpp/experimental/fs/last_write_time.

```

std::time_t cftime = std::chrono::system_clock::to_time_t(sTime);
std::cout << "Write time on server " // (3)
          << std::asctime(std::localtime(&cftime));
std::cout << "Write time on server " // (4)
          << std::asctime(std::gmtime(&cftime)) << '\n';

const auto fTime2 = fTime + 2h; // (5)
const auto sTime2 = std::chrono::clock_cast<std::chrono::system_
clock>(fTime2);
// (6)

std::time_t cftime2 = std::chrono::system_clock::to_time_t(sTime2);
std::cout << "Local time on client "
          << std::asctime(std::localtime(&cftime2)) << '\n';

```

В строке (1) указано время записи только что созданного файла. В строке (2) используется переменная `fTime`, чтобы выполнить приведение времени в реально-временные настенные часы `sTime`. В следующей строке в переменную `sTime` инициализируется экземпляр класса `std::chrono::system_clock`. Объект `cftime` имеет тип `std::filesystem::file_time_type`, который в данном случае является псевдонимом класса `std::chrono::system_clock`; поэтому в строке (3) можно инициализировать местное время `std::localtime` и выразить календарное время в текстовом представлении. Если вместо местного времени `std::localtime` использовать всемирное координированное время (UTC) `std::gmtime` (4), то ничего не изменится, что озадачивает, потому что всемирное координированное время отличается от местного времени Германии на 2 часа. Это связано с сервером онлайн-компилятора на веб-сайте `en.cppreference.com`. На сервере всемирное координированное время и местное время настроены на одно и то же время.

Ниже приведен результат работы программы. Время записи файла было сдвинуто на 2 часа в будущее (5), и затем файл был прочитан обратно из файловой системы (6). В результате происходит корректировка времени, чтобы оно соответствовало местному времени Германии.

```

Write time on server Tue Oct 10 06:28:04 2017
Write time on server Tue Oct 10 06:28:04 2017

Local time on client Tue Oct 10 08:28:04 2017

```

Информация о пространстве в файловой системе

Вспомогательная функция `std::filesystem::space` возвращает объект `std::filesystem::space_info` с тремя членами: `capacity`, `free` и `available`.

- `capacity` (емкость) – это суммарный размер файловой системы.
- `free` (свободно) – это свободное пространство в файловой системе.

- available (доступно) – это свободное пространство для непривилегированного процесса (равное или меньше свободному).

Все размеры указываются в байтах.

Результаты работы следующей ниже программы взяты с веб-сайта en.cppreference.com. Все опробованные пути находились в одной и той же файловой системе; поэтому всегда получался один и тот же ответ.

Информация о пространстве

```
// space.cpp
...
#include <filesystem>

...

namespace fs = std::filesystem;

...

fs::space_info root = fs::space("/");
fs::space_info usr = fs::space("/usr");

std::cout << ". Capacity Free Available\n"
  << "/" << root.capacity << " "
  << root.free << " " << root.available << "\n"
  << "usr " << usr.capacity << " "
  << usr.free << " " << usr.available;
```

	Capacity	Free	Available
/	42140499968	18342744064	17054289920
usr	42140499968	18342744064	17054289920

Типы файлов

Используя следующие ниже предикаты, можно легко запрашивать тип файла.

Таблица. Типы файлов в файловой системе

Типы файлов	Описание
<code>is_block_file</code>	Проверяет, не относится ли путь к блочному файлу
<code>is_character_file</code>	Проверяет, не ссылается ли путь на символьный файл
<code>is_directory</code>	Проверяет, не ссылается ли путь на каталог

Типы файлов	Описание
<code>is_empty</code>	Проверяет, не ссылается ли путь на пустой файл или каталог
<code>is_fifo</code>	Проверяет, не ссылается ли путь на именованный конвейер (канал) ¹
<code>is_other</code>	Проверяет, не ссылается ли путь на другой файл
<code>is_regular_file</code>	Проверяет, не ссылается ли путь на обычный файл
<code>is_socket</code>	Проверяет, не ссылается ли путь на IPC-сокеты
<code>is_symlink</code>	Проверяет, не ссылается ли путь на символическую ссылку
<code>status_known</code>	Проверяет, есть ли сведения о статусе файла

Получение типа файла

Указанные выше предикаты предоставляют информацию о типе файла. Для одного файла может действовать более одного предиката. Символическая ссылка, ссылающаяся на обычный файл, является одновременно и обычным файлом, и символической ссылкой.

Тип файла

```
// fileType.cpp
...
#include <filesystem>

...

namespace fs = std::filesystem;

void printStatus(const fs::path& path_){
    std::cout << path_;
    if(!fs::exists(path_)) std::cout << " does not exist";
    else{
        if(fs::is_block_file(path_) std::cout << " is a block file\n";
        if(fs::is_character_file(path_) std::cout << " is a character device\n";
        if(fs::is_directory(path_) std::cout << " is a directory\n";
        if(fs::is_fifo(path_) std::cout << " is a named pipe\n";
        if(fs::is_regular_file(path_) std::cout << " is a regular file\n";
        if(fs::is_socket(path_) std::cout << " is a socket\n";
        if(fs::is_symlink(path_) std::cout << " is a symlink\n";
    }
}

...
```

¹ См. https://en.wikipedia.org/wiki/Named_pipe.

```
fs::create_directory("rainer");
printStats("rainer");

std::ofstream("rainer/regularFile.txt");
printStats("rainer/regularFile.txt");

fs::create_directory("rainer/directory");
printStats("rainer/directory");

mkfifo("rainer/namedPipe", 0644);
printStats("rainer/namedPipe");

struct sockaddr_un addr;
addr.sun_family = AF_UNIX;
std::strcpy(addr.sun_path, "rainer/socket");
int fd = socket(PF_UNIX, SOCK_STREAM, 0);
bind(fd, (struct sockaddr*)&addr, sizeof addr);
printStats("rainer/socket");

fs::create_symlink("rainer/regularFile.txt", "symlink");
printStats("symlink");

printStats("dummy.txt");

fs::remove_all("rainer");
```

```
"rainer" is a directory
"rainer/regularFile.txt" is a regular file
"rainer/directory" is a directory
"rainer/namedPipe" is a named pipe
"rainer/socket" is a socket
"symlink" is a regular file
           is a symlink
"dummy.txt" does not exist
```

19. Многопоточность



Циппи заплетает косу

Краткий обзор

Начиная со стандарта C++11 многопоточность в языке C++ поддерживается нативно. Новый поток инструкций начинает исполняться немедленно и может работать как на переднем, так и на заднем плане. Для управления доступом к коллективным переменным между потоками используются механизмы синхронизации, такие как мьютексы или блокировочные замки.

Описанная выше поддержка состоит из двух частей: четко определенной модели памяти и стандартизированного интерфейса многопоточности.

Модель памяти

Основой многопоточности является четко определенная модель памяти. Данная модель должна учитывать следующие ниже моменты:

- атомарные операции: операции, которые могут выполняться без прерывания;
- частичное упорядочивание операций: последовательность операций, которые не должны переупорядочиваться;
- видимые эффекты операций: гарантии того, что операции на коллективных переменных будут видны другим потокам инструкций.

Модель памяти C++ имеет много общего со своей предшественницей – моделью памяти Java. В отличие от этой модели, язык C++ позволяет нарушать последовательную состыкованность. Поведение атомарных операций по умолчанию подчиняется принципу последовательной состыкованности.

Последовательная состыкованность обеспечивает две гарантии.

1. Инструкции программы исполняются в порядке следования исходного кода.
2. Во всех потоках инструкций существует глобальный порядок следования всех операций.

Атомарные типы данных

В языке C++ есть набор атомарных типов данных. Во-первых, это низкоуровневый конкретный класс `std::atomic_flag`, а во-вторых, шаблонный класс `std::atomic`. Кроме того, с помощью шаблонного класса `std::atomic` можно определять свой собственный атомарный тип данных.

`std::atomic_flag`

Конкретный класс `std::atomic_flag` представляет атомарную булеву величину. У него есть состояние `clear` и `set`. Ради простоты состояние `clear` будем называть ложным (`false`), а состояние `set` – истинным (`true`). Функция-член `clear` позволяет устанавливать его значение равным `false`. С помощью функций-членов `test_and_set` можно возвращать значение `true` и возвращать предыдущее значение. Функция-член, которая могла бы запрашивать текущее значение, отсутствует. Это изменилось с появлением стандарта C++20. В стандарте C++20 конкретный класс `std::atomic_flag` имеет функцию-член `test` и может использоваться для синхронизации потоков с помощью функций-членов `notify_one`, `notify_all` и `wait`.

Таблица. Все операции на объекте `std::atomic_flag atomicFlag`

Функция-член	Описание
<code>atomicFlag.clear()</code>	Очищает атомарный флаг
<code>atomicFlag.test_and_set()</code>	Устанавливает атомарный флаг и возвращает старое значение
<code>atomicFlag.test()</code> (C++20)	Возвращает значение флага
<code>atomicFlag.notify_one()</code> (C++20)	Уведомляет один поток, ожидающий атомарный флаг
<code>atomicFlag.notify_all()</code> (C++20)	Уведомляет все потоки, ожидающие атомарный флаг
<code>atomicFlag.wait(b)</code> (C++20)	Блокирует поток до тех пор, пока не получит уведомление и атомарное значение не изменится

Вызов функции-члена `atomicFlag.test()` возвращает значение объекта `atomicFlag` без его изменения. Далее объект `std::atomic_flag atomicFlag` мож-

но использовать для синхронизации потоков: `atomicFlag.wait()`, `atomicFlag.notify_one()` и `atomicFlag.notify_all()`. Функции-члены `notify_one` или `notify_all` уведомляют один или все ожидающие атомарные флаги. Функция-член `atomicFlag.wait(boo)` нуждается в булевой величине `b`. Вызов `atomicFlag.wait(b)` блокирует до тех пор, пока не получит уведомление. Она выполняет проверку на равенство значения объекта `atomicFlag` значению `b` и разблокирует, если они не равны.

Конкретный класс `std::atomic_flag` необходимо инициализировать явным образом, используя `ATOMIC_FLAG_INIT`: `std::atomic_flag flag(ATOMIC_FLAG_INIT)`. В стандарте C++20 при дефолтном конструировании экземпляра класса `std::atomic_flag` он будет находиться в состоянии `false`.

Замечательным свойством конкретного класса `std::atomic_flag` является то, что это единственный атомарный тип, который гарантированно свободен от блокировочных замков¹. Остальные мощные атомарные типы могут обеспечивать свою функциональность, используя механизм установки и снятия блокировочных замков, такой как мьютекс `std::mutex`.

std::atomic

У шаблонного класса `std::atomic` есть различные конкретизации. Для них требуется заголовок `<atomic>`. Конкретизированные классы `std::atomic<bool>` и `std::atomic<user-defined type>` используют первичный шаблонный класс. Частичные конкретизации доступны для указателей `std::atomic<T*>`, а начиная со стандарта C++20 – и для умных указателей `std::atomic<smart T*>`; полные конкретизированные классы доступны для интегральных типов `std::atomic<integral type>`, а начиная со стандарта C++20 – и для типов с плавающей точкой `std::atomic<floating-point>`. С помощью шаблонного класса `std::atomic` можно определять свой собственный атомарный тип данных.

Фундаментальный интерфейс атомарных типов

Три частично конкретизированных класса `std::atomic<bool>`, `std::atomic<user-defined type>` и `std::atomic<smart T*>` поддерживают фундаментальный интерфейс атомарных типов.

Функция-член	Описание
<code>is_lock_free</code>	Проверяет, свободен ли атомарный объект от блокировочного замка
<code>atomic_ref<T>::is_always_lock_free</code>	Проверяет во время компиляции, всегда ли атомарный тип свободен от блокировочного замка
<code>load</code>	Атомарно возвращает значение атомарного объекта
<code>operator T</code>	Атомарно возвращает значение атомарного объекта. Эквивалентно вызову функции-члена <code>atom.load()</code>

¹ См. https://en.wikipedia.org/wiki/Non-blocking_algorithm.

Функция-член	Описание
<code>store</code>	Атомарно заменяет атомарное значение на неатомарное
<code>exchange</code>	Атомарно заменяет значение на новое. Возвращает старое значение
<code>compare_exchange_strong</code> <code>compare_exchange_weak</code>	Атомарно сравнивает и в итоге обменивает значение
<code>notify_one</code> (C++20)	Уведомляет об одной атомарной операции ожидания
<code>notify_all</code> (C++20)	Уведомляет обо всех атомарных операциях ожидания
<code>wait</code> (C++20)	Блокирует до тех пор, пока не получит уведомление. Если старое значение изменилось, то возвращает его

Атомарная операция `compare_exchange_strong` имеет следующий синтаксис: `bool compare_exchange_strong(T& expected, T& desired)`. Ее поведение описывается следующим образом:

- если атомарное сравнение объекта `atomicValue` со значением объекта `expected` возвращает `true`, то значение объекта `atomicValue` в той же атомарной операции устанавливается равным `desired`;
- если сравнение возвращает `false`, то значение объекта `expected` устанавливается равным `atomicValue`.

Пользовательские атомарные типы `std::atomic<user-defined type>`

Благодаря шаблонному классу `std::atomic` можно определять свой собственный пользовательский атомарный тип.

При определении пользовательского атомарного типа `std::atomic<user-defined type>` существует ряд существенных ограничений на пользовательский тип. Атомарный тип `std::atomic<user-defined type>` поддерживает тот же интерфейс, что у специального конкретизированного шаблонного класса `std::atomic<bool>`.

Ниже перечислены ограничения, накладываемые на пользовательский тип, при его определении как атомарного:

- оператор копирующего присваивания у пользовательского типа должен быть тривиальным для всех его базовых классов и нестатических членов. Нет возможности определить оператор копирующего присваивания, однако его можно запрашивать у компилятора с помощью ключевого слова `default`¹;
- пользовательский тип не должен иметь ни виртуальных функций-членов, ни виртуальных базовых классов;

¹ См. <http://en.cppreference.com/w/cpp/keyword/default>.

- пользовательский тип должен допускать побитовое сравнение, чтобы обеспечивать возможность применения функций `C memcpu`¹ или `memcmp`².

Большинство популярных платформ допускают использование атомарных операций для пользовательских атомарных типов `std::atomic<user-defined type>`, если размер пользовательского типа не превышает `int`.

Атомарные умные указатели `std::atomic<smart T*>` (C++20)

Коллективный умный указатель `std::shared_ptr` состоит из управляющего блока и его ресурса. Управляющий блок является потокобезопасным, а доступ к ресурсу – нет, в силу чего модификация счетчика ссылок является атомарной операцией, и есть гарантия того, что ресурс будет удален только один раз. Именно такие гарантии дает коллективный умный указатель `std::shared_ptr`. Использование частичной конкретизации `std::atomic<std::shared_ptr<T>>` и `std::atomic<std::weak_ptr<T>>` шаблонного класса дает дополнительную гарантию того, что доступ к базисному объекту является потокобезопасным. В стандарте 2022 все операции на атомарных умных указателях не свободны от блокировочных замков.

`std::atomic<floating-point type>` (C++20)

Дополнительно к фундаментальному интерфейсу атомарных типов атомарный конкретизированный тип `std::atomic<floating-point type>` поддерживает сложение и вычитание.

Таблица. Операции в дополнение к фундаментальному интерфейсу атомарных типов

Функция-член	Описание
<code>fetch_add, +=</code>	Атомарно прибавляет (вычитает) значение
<code>fetch_sub, -=</code>	Возвращает старое значение

Для типов `float`, `double` и `long double` имеются полные конкретизации.

`std::atomic<T*>`

Класс `std::atomic<T*>` – это частичная конкретизация шаблонного класса `std::atomic`. Он ведет себя как обычный указатель `T*`. Дополнительно к атомарному конкретизированному классу `std::atomic<floating-point type>` конкретизированный класс `std::atomic<T*>` поддерживает операции предварительного и последующего увеличения или предварительного и последующего уменьшения.

Таблица. Операции в дополнение к атомарному конкретизированному классу `std::atomic<floating-point type>`

Функция-член	Описание
<code>++, --</code>	Увеличивает или уменьшает (до и после) атомарный объект

¹ См. <http://en.cppreference.com/w/cpp/string/byte/memcpu>.

² См. <http://en.cppreference.com/w/cpp/string/byte/memcmp>.

Давайте взглянем на короткий пример.

```
int intArray[5];
std::atomic<int*> p(intArray);
p++;
assert(p.load() == &intArray[1]);
p+=1;
assert(p.load() == &intArray[2]);
--p;
assert(p.load() == &intArray[1]);
```

std::atomic<integral type>

Для каждого интегрального типа существует полный конкретизированный класс `std::atomic<integral type>` шаблонного класса `std::atomic`.

Конкретизированный класс `std::atomic<integral type>` поддерживает все операции, которые поддерживаются конкретизированным классом `std::atomic<T*>` или `std::atomic<floating-point type>` шаблонного класса. Кроме того, конкретизированный класс `std::atomic<integral type>` поддерживает побитовые логические операторы AND, OR и XOR.

Таблица. Все операции на атомарных объектах

Функция-член	Описание
<code>fetch_or, =</code>	Атомарно выполняет побитовые операции (AND, OR и XOR) со значением
<code>fetch_and, &=</code>	Возвращает старое значение
<code>fetch_xor, ^=</code>	

Между составной операцией побитового присваивания и вариантом с доставкой существует небольшое различие. Составная операция побитового присваивания возвращает новое значение, а вариант с доставкой – старое.

При более глубоком рассмотрении выясняется следующее: операции атомарного умножения, атомарного деления и атомарного сдвига не доступны. Это не является существенным ограничением, поскольку такие операции требуются редко и могут быть легко реализованы. Ниже приведен пример атомарной функции `fetch_mult`.

Атомарное умножение с использованием функции-члена `compare_exchange_strong`

```
// fetch_mult.cpp

#include <atomic>
#include <iostream>

template <typename T>
T fetch_mult(std::atomic<T>& shared, T mult){
    T oldValue = shared.load();
```

```

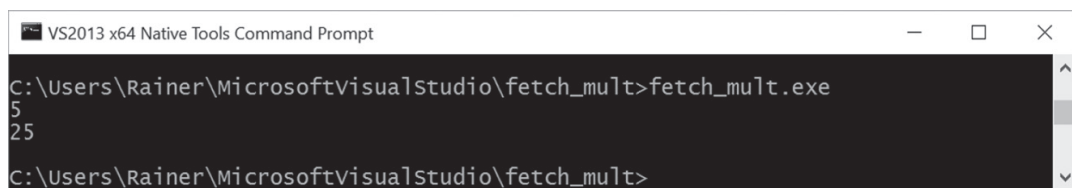
while (!shared.compare_exchange_strong(oldValue, oldValue * mult));
return oldValue;
}

int main(){
    std::atomic<int> myInt{5};
    std::cout << myInt << '\n';
    fetch_mult(myInt,5);
    std::cout << myInt << '\n';
}

```

Здесь стоит отметить, что умножение в строке 9 происходит только в том случае, если соблюдается отношение `oldValue == shared`. Умножение помещено в цикл `while`, чтобы быть уверенным в том, что умножение всегда происходит, потому что имеются две инструкции для чтения значения переменной `oldValue` в строке 8 и его использования в строке 9.

Ниже приведен результат атомарного умножения.



```

VS2013 x64 Native Tools Command Prompt
C:\Users\Rainer\MicrosoftVisualStudio\fetch_mult>fetch_mult.exe
5
25
C:\Users\Rainer\MicrosoftVisualStudio\fetch_mult>

```

Атомарное умножение

std::atomic_ref

Шаблонный класс `std::atomic_ref` применяет атомарные операции к объекту, на который ссылается. Конкурентная запись и чтение атомарных объектов осуществляются потокобезопасным образом. Время жизни объекта, на который имеются ссылки, должно превышать время жизни объекта класса `std::atomic_ref`. Шаблонный класс `std::atomic_ref` поддерживает те же типы и операции, как это делает шаблонный класс `std::atomic` для своего базисного типа.

```

struct Counters {
    int a;
    int b;
};

Counter counter;
std::atomic_ref<Counters> cnt(counter);

```

Все атомарные операции

В следующей ниже таблице представлены все атомарные операции. Для получения дополнительной информации об операциях следует обратиться к предыдущим разделам об атомарных типах данных.

Таблица. Все атомарные операции в зависимости от атомарного типа

Функция-член	atomic_flag	atomic<bool> atomic<user> atomic<smart T*>	atomic<float>	atomic<T*>	atomic<int>
test_and_set	да				
clear	да				
is_lock_free		да	да	да	да
atomic<T>::is_always_lock_free		да	да	да	да
load		да	да	да	да
operator T		да	да	да	да
store		да	да	да	да
exchange		да	да	да	да
compare_exchange_strong		да	да	да	да
compare_exchange_weak		да	да	да	да
fetch_add, +=			да	да	да
fetch_sub, -=			да	да	да
fetch_or, =					да
fetch_or, =					да
fetch_xor, ^=					да
++, --				да	да
notify_one (C++20)	да	да	да	да	да
notify_all (C++20)	да	да	да	да	да
wait (C++20)	да	да	да	да	да

Конкретизированный класс `std::atomic<float>` обозначает атомарные типы с плавающей точкой¹, а конкретизированный класс `std::atomic<int>` – атомарные интегральные типы².

¹ См. https://en.cppreference.com/w/cpp/types/is_floating_point.

² См. https://en.cppreference.com/w/cpp/types/is_integral.

Потоки инструкций

Для использования интерфейса многопоточности языка C++ нужен заголовок `<thread>`. Язык C++ располагает классом `std::thread` стандарта C++11 и улучшенным классом `std::jthread` стандарта C++20.

`std::thread`

Создание

Класс `std::thread` представляет исполняемую единицу кода. Эта исполняемая единица кода, которую поток немедленно запускает, получает свой рабочий пакет в качестве вызываемой единицы кода. Вызываемая единица кода может быть функцией, функциональным объектом либо лямбда-функцией:

Создание потока инструкций

```
// threadCreate.cpp
...
#include <thread>

...

using namespace std;

void helloFunction(){
    cout << "function" << endl;
}

class HelloFunctionObject {
public:
    void operator()() const {
        cout << "function object" << endl;
    }
};

thread t1(helloFunction); // function

HelloFunctionObject helloFunctionObject;
thread t2(helloFunctionObject); // function object

thread t3([]{ cout << "lambda function"; }); // lambda function
```

Время жизни

Создатель потока инструкций должен позаботиться о времени жизни создаваемого им потока. Исполняемая единица кода созданного потока инструкций заканчивается с окончанием вызываемой единицы кода. Создатель либо ждет до тех пор, пока созданный поток `t` не завершится: `t.join()`, либо отсоединяется от созданного потока: `t.detach()`. К потоку `t` можно присоединиться, если на нем не было выполнено ни одного вызова `t.join()` или `t.detach()`. Пригодный для присоединения поток генерирует в своем деструкторе функцию немедленного завершения `std::terminate`, и программа завершается.



Потоки инструкций рекомендуется перемещать с осторожностью

Потоки инструкций можно перемещать, но не копировать.

```
#include <thread>
...

std::thread t([]{ cout << "lambda function"; });
std::thread t2;
t2 = std::move(t);

std::thread t3([]{ cout << "lambda function"; });
t2 = std::move(t3); // std::terminate
```

При выполнении присваивания `t2 = std::move(t)` поток инструкций `t2` получает вызываемую единицу кода от потока инструкций `t`. Если допустить, что поток `t2` уже имел вызываемую единицу кода и пригоден для присоединения, то среда исполнения C++ вызовет функцию немедленного завершения `std::terminate`. Это происходит именно в присваивании `t2 = std::move(t3)`, поскольку поток `t2` ранее не исполнял ни `t2.join()`, ни `t2.detach()`.

Время жизни потока инструкций

```
// threadLifetime.cpp
...
#include <thread>
...

thread t1(helloFunction); // function

HelloFunct
ionObject helloFunctionObject;
thread t2(helloFunctionObject); // function object
```

```
thread t3([]{ cout << "lambda function"; }); // lambda function

t1.join();
t2.join();
t3.join();
```

Отсоединенный от своего создателя поток инструкций обычно принято называть потоком-демоном, поскольку он работает в фоновом режиме.

Аргументы

Шаблонный класс `std::thread` является вариативным, в силу чего он может получать произвольное количество аргументов по копии либо по ссылке. Получать аргументы может либо вызываемая единица кода, либо поток инструкций. Поток инструкций делегирует их вызываемым единицам кода: `tPerCopy2` либо `tPerReference2`.

```
#include <thread>

...

using namespace std;

void printStringCopy(string s){ cout << s; }
void printStringRef(const string& s){ cout << s; }

string s{"C++"};

thread tPerCopy([=]{ cout << s; });           // C++
thread tPerCopy2(printStringCopy, s);        // C++
tPerCopy.join();
tPerCopy2.join();

thread tPerReference([&]{ cout << s; });     // C++
thread tPerReference2(printStringRef, s);    // C++
tPerReference.join();
tPerReference2.join();
```

Первые два потока инструкций получают свой аргумент `s` по копии, вторые два – по ссылке.



По умолчанию потоки инструкций должны получать свои аргументы по копии

Аргументы потока инструкций

```
// threadArguments.cpp
...
#include <thread>

...

using std::this_thread::sleep_for;
using std::this_thread::get_id;

struct Sleeper{
    Sleeper(int& i_):i{i_}{};
    void operator() (int k){
        for (unsigned int j= 0; j <= 5; ++j){
            sleep_for(std::chrono::milliseconds(100));
            i += k;
        }
        std::cout << get_id(); // неопределенное поведение
    }
private:
    int& i;
};

int valSleeper= 1000;

std::thread t(Sleeper(valSleeper), 5);
t.detach();

std::cout << valSleeper; // неопределенное поведение
```

Приведенный выше фрагмент программы имеет неопределенное поведение. Во-первых, время жизни потока вывода данных `std::cout` привязано ко времени жизни главного потока инструкций, а созданный поток инструкций получает свою переменную `valSleeper` по ссылке. Проблема в том, что созданный поток инструкций может жить дольше своего создателя, поэтому поток вывода данных `std::cout` и переменная `valSleeper` теряют свою допустимость, если главный поток инструкций завершает свою работу. Во-вторых, `valSleeper` – это коллективная, мутируемая переменная, которая конкурентно используется главным и дочерним потоками инструкций. Следовательно, возникает гонка за данными.

Операции

На потоке инструкций `t` можно выполнять многочисленные операции.

Таблица. Операции на объекте `t` шаблонного класса `std::thread`

Функция-член	Описание
<code>t.join()</code>	Ожидает завершения потока инструкций, т. е. до тех пор, пока поток инструкций <code>t</code> не завершит свою исполняемую единицу кода
<code>t.detach()</code>	Исполняет созданный поток инструкций <code>t</code> независимо от создателя
<code>t.joinable()</code>	Проверяет, поддерживает ли поток инструкций <code>t</code> вызовы функций-членов <code>join</code> или <code>detach</code>
<code>t.get_id()</code> и <code>std::this_thread::get_id()</code>	Возвращает идентичность потока инструкций
<code>std::thread::hardware_concurrency()</code>	Указывает количество потоков инструкций, которые могут работать параллельно
<code>std::this_thread::sleep_until(absTime)</code>	Помещает поток инструкций <code>t</code> в сон до тех пор, пока не наступит время <code>absTime</code>
<code>std::this_thread::sleep_for(relTime)</code>	Помещает поток инструкций <code>t</code> в сон на время <code>relTime</code>
<code>std::this_thread::yield()</code>	Предлагает системе запустить еще один поток инструкций
<code>t.swap(t2)</code> и <code>std::swap(t1, t2)</code>	Меняет потоки инструкций местами

Функции-члены `t.join()` или `t.detach()` можно вызывать на потоке `t` только один раз. Если попытаться вызвать их более одного раза, то будет сгенерировано исключение `std::system_error`. Статический метод `std::thread::hardware_concurrency` шаблонного класса `std::thread` возвращает количество ядер либо 0, если среда исполнения не может определить это количество. Для операций `sleep_until` и `sleep_for` в качестве аргумента требуется временная точка либо продолжительность времени.

Потоки инструкций нельзя копировать, но можно перемещать. Операция `swap` выполняет перемещение, если это возможно.

Операции на потоке инструкций

```
// threadMember Functions.cpp
...
#include <thread>
...

using std::this_thread::get_id;
```

```
std::thread::hardware_concurrency(); // 4

std::thread t1([]{ get_id(); }); // 139783038650112
std::thread t2([]{ get_id(); }); // 139783030257408
t1.get_id(); // 139783038650112
t2.get_id(); // 139783030257408

t1.swap(t2);

t1.get_id(); // 139783030257408
t2.get_id(); // 139783038650112
get_id(); // 140159896602432
```

std::jthread

Название конкретного класса `std::jthread` переводится с англ. как присоединяющийся поток инструкций (joining thread). Дополнительно к классу `std::thread` из стандарта C++11 класс `std::jthread` может автоматически присоединяться к запущенному потоку инструкций и сигнализировать о прерывании.

Автоматическое присоединение

Неинтуитивное поведение потока инструкций `std::thread` заключается в следующем: если поток `std::thread` по-прежнему допускает присоединение, то в его деструкторе вызывается функция немедленного завершения `std::terminate`.

В противоположность этому поток `thr` как экземпляр конкретного класса `std::jthread` присоединяется в своем деструкторе автоматически, если поток `thr` по-прежнему допускает присоединение.

Завершение потока `std::jthread`, который по-прежнему допускает присоединение

```
// jthreadJoinable.cpp
...
#include <thread>

...

std::jthread thr([]{ std::cout << "std::jthread" << "\n"; }); // std::jthread

std::cout << "thr.joinable(): " << thr.joinable() << "\n"; // thr.joinable(): true
```

В отличие от потока `std::thread`, поток `std::jthread` допускает прерывание, поддерживая сигналы остановки, что позволяет безопасно останавливать поток, если это необходимо.

Сигнал остановки

Дополнительная функциональность кооперативного присоединения потока инструкций основана на классах `std::stop_token`, `std::stop_callback` и `std::stop_source`, введенных в стандарт C++20. Следующая ниже программа должна дать общее представление об их работе.

Прерывание непрерываемого и прерываемого потоков `std::jthread`

```
// interruptJthread.cpp
...

#include <thread>
#include <stop_token>

using namespace::std::literals;

...

std::jthread nonInterruptable([]{ // (1)
    int counter{0};
    while (counter < 10){
        std::this_thread::sleep_for(0.2s);
        std::cerr << "nonInterruptable: " << counter << '\n';
        ++counter;
    }
});

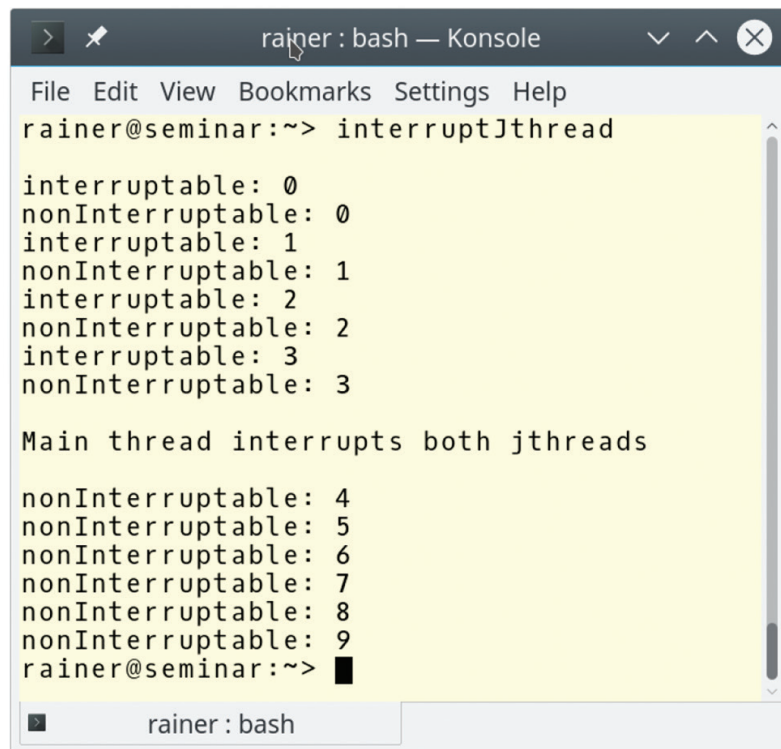
std::jthread interruptable([](std::stop_token token){ // (2)
    int counter{0};
    while (counter < 10){
        std::this_thread::sleep_for(0.2s);
        if (token.stop_requested()) return; // (3)
        std::cerr << "interruptable: " << counter << '\n';
        ++counter;
    }
});

std::this_thread::sleep_for(1s);

std::cerr << "Main thread interrupts both jthreads" << std::endl;
nonInterruptable.request_stop(); // (4)
interruptable.request_stop();
```

В главной программе запускается два потока инструкций – непрерываемый и прерываемый: (1) и (2). В отличие от непрерываемого потока `nonInterruptable`,

прерываемый поток `interruptable` получает сигнал остановки `std::stop_token` и использует его в строке (3) для проверки, не был ли он прерван: `token.stop_requested()`. В случае прерывания лямбда-функция возвращается, и поток инструкций завершается. Вызов функции-члена `interruptable.request_stop()` (4) инициирует завершение потока инструкций. Это не относится к предыдущему вызову функции-члена `nonInterruptable.request_stop()`, который не имеет эффекта.



```

rainer@seminar:~> interruptJthread

interruptable: 0
nonInterruptable: 0
interruptable: 1
nonInterruptable: 1
interruptable: 2
nonInterruptable: 2
interruptable: 3
nonInterruptable: 3

Main thread interrupts both jthreads

nonInterruptable: 4
nonInterruptable: 5
nonInterruptable: 6
nonInterruptable: 7
nonInterruptable: 8
nonInterruptable: 9
rainer@seminar:~>

```

`std::stop_token`, `std::stop_source` и `std::stop_callback`

Объекты классов `stop_token`, `std::stop_callback` или `std::stop_source` позволяют асинхронно запрашивать остановку исполнения или спрашивать, получило ли исполнение сигнал остановки. Сигнал остановки `std::stop_token` может передаваться в операции и затем использоваться для активного опрашивания сигнала, может ли сигнал остановки быть остановлен, либо для регистрации функции обратного вызова через объект класса `std::stop_callback`. Объект-источник остановки `std::stop_source` посылает запрос на остановку. Этот запрос влияет на все ассоциированные сигналы остановки `std::stop_token`. Три объекта `std::stop_source`, `std::stop_token` и `std::stop_callback` коллективно владеют ассоциированным состоянием остановки. При вызове функций-членов `request_stop()`, `stop_requested()` и `stop_possible()` выполняются атомарные операции.

Конструируемый по умолчанию объект класса `std::stop_source` инициализируется как источник остановки. Конструктор, принимающий аргумент `std::nostopstate_t`, строит пустой объект `std::stop_source` без ассоциированного с ним состояния остановки.

Конструкторы класса `std::stop_source`

```
stop_source();
explicit stop_source(std::nostopstate_t) noexcept;
```

Классы `std::stop_source` и `std::stop_token` предоставляют следующие ниже функции-члены для обработки остановок.

Таблица. Функции-члены объекта `src` класса `std::stop_source`

Функция-член	Описание
<code>src.get_token()</code>	Если остановка возможна (<code>stop_possible()</code>), то возвращает сигнал остановки <code>stop_token</code> для ассоциированного с ним состояния остановки. В противном случае возвращает сконструированный по умолчанию (пустой) сигнал остановки <code>stop_token</code>
<code>src.stop_possible()</code>	Возвращает <code>true</code> , если объект <code>src</code> может быть запрошен на предмет остановки
<code>src.stop_requested()</code>	Возвращает <code>true</code> , если функции-члены <code>stop_possible()</code> и <code>request_stop()</code> были вызваны одним из владельцев
<code>src.request_stop()</code>	Вызывает запрос на остановку, если остановка возможна (<code>stop_possible()</code>) и запрос на остановку не делался (<code>!stop_requested()</code>). В противном случае вызов не имеет никакого эффекта

Функция-член `src.stop_possible()` означает, что с объектом `src` ассоциировано состояние остановки. Функция-член `src.stop_requested()` возвращает `true`, если с объектом `src` ассоциировано состояние остановки и ранее был запрос на остановку. Функция-член `src.request_stop()` является успешной и возвращает `true`, если с объектом `src` ассоциировано состояние остановки и был получен запрос на остановку.

Вызов функции-члена `src.get_token()` возвращает сигнал остановки `token`. Благодаря сигналу остановки `token` можно проверять, был ли сделан или возможно сделан запрос на остановку для ассоциированного с ним объекта-источника остановки `src`. Сигнал остановки `token` наблюдает за объектом-источником остановки `src`.

Таблица. Функции-члены объекта `token` класса `std::stop_token`

Функция-член	Описание
<code>token.stop_possible()</code>	Возвращает <code>true</code> , если с сигналом остановки <code>token</code> ассоциировано состояние остановки
<code>token.stop_requested()</code>	Возвращает <code>true</code> , если функция-член <code>request_stop()</code> была вызвана на ассоциированном объекте-источнике остановки <code>std::stop_source src</code> , иначе возвращает <code>false</code>

Со сконструированным по умолчанию сигналом остановки состояние остановки не ассоциировано. Функция-член `token.stop_possible` также возвраща-

ет true, если запрос на остановку уже был сделан. Функция-член `token.stop_requested()` возвращает true, если с сигналом остановки ассоциировано состояние остановки и он уже получил запрос на остановку.

Если сигнал остановки `std::stop_token` должен быть временно отключен, то его можно заменить сигналом, сконструированным по умолчанию. В следующем ниже фрагменте исходного кода показано, как отключать и включать способность потока инструкций принимать запросы на остановку.

Временное отключение сигнала остановки

```
std::jthread jthr([](std::stop_token token) {
    ...
    std::stop_token interruptDisabled;
    std::swap(token, interruptDisabled); // (1)
    ...                               // (2)
    std::swap(token, interruptDisabled);
    ...
})
```

С сигналом остановки `std::stop_token interruptDisabled` состояние остановки не ассоциировано, в силу чего поток инструкций `jthr` может принимать запросы на остановку во всех строках, кроме строк (1) и (2).

Коллективные переменные

Если переменная используется несколькими потоками инструкций коллективно, то доступ к ней необходимо координировать. В языке C++ этим занимаются мьютексы и блокировочные замки.

Гонка за данными

Гонка за данными – это состояние, при котором как минимум два потока инструкций обращаются к коллективным данным одновременно и как минимум один из потоков инструкций является пишущим, и поэтому программа имеет неопределенное поведение.

Чередование потоков инструкций можно хорошо наблюдать в ситуациях, когда несколько потоков инструкций пишут в поток вывода данных `std::cout`. В этом случае поток вывода данных `std::cout` является коллективной переменной.

Несинхронизированная запись в поток вывода данных `std::cout`

```
// withoutMutex.cpp
...
#include <thread>
```

...

```
using namespace std;
```

```
struct Worker{
    Worker(string n):name(n){};
    void operator() (){
        for (int i= 1; i <= 3; ++i){
            this_thread::sleep_for(chrono::milliseconds(200));
            cout << name << ": " << "Work " << i << endl;
        }
    }
private:
    string name;
};
```

```
thread herb= thread(Worker("Herb"));
thread andrei= thread(Worker(" Andrei"));
thread scott= thread(Worker (" Scott"));
thread bjarne= thread(Worker(" Bjarne"));
```

```
rainer: bash - Konsole
Datei Bearbeiten Ansicht Lesezeichen Einstellungen Hilfe
rainer@icho:~> writeToCout
Andrei Scott: : Work Work 11
Bjarne: Work 1
Herb: Work 1
Andrei: BjarneWork : Work Scott2: 2Work
Herb: 2Work 2
Andrei: Work 3
Scott: Work Herb: Work 33
Bjarne: Work 3
rainer@icho:~> █
```

Запись в поток вывода данных `std::cout` не скоординирована.



Потоки данных являются потокобезопасными

Стандарт C++11 гарантирует, что символы пишутся атомарно. Следовательно, их не нужно защищать. Защита чередования потоков инструкций на потоке данных требуется только в том случае, если вся операция чтения целиком не чередуется. Эта гарантия соблюдается для потоков ввода и вывода данных.

В стандарте C++20 появились синхронизированные потоки вывода данных, такие как `std::osyncstream` и `std::wosyncstream`. Они гарантируют, что запись в один поток вывода данных будет синхронизирована. Вывод пишется во внутренний буфер и сбрасывается, когда он выходит за пределы области видимости. Синхронизированный поток вывода данных может иметь имя, например `synced_out`, либо его не иметь.

Синхронизированные потоки вывода данных

```
{
    std::osyncstream synced_out(std::cout);
    synced_out << "Hello, ";
    synced_out << "World!";
    synced_out << '\n';           // эффект отсутствует
    synced_out << "and more!\n";
} // разрушает synced_output и эмитирует внутренний буфер

std::osyncstream(std::cout) << "Hello, " << "World!" << "\n";
```

В приведенном выше примере поток вывода данных `std::cout` является коллективной переменной, которая должна иметь исключительный доступ к потоку данных.

Мьютексы

Мьютекс (от англ. mutual exclusion, то есть взаимное исключение) — как объект особого класса `std::mutex` гарантирует, что только один поток инструкций может получать доступ к критической области в одно и то же время. Для него нужен заголовок `<mutex>`. Мьютекс `m` запирает критическую секцию вызовом функции-члена `m.lock()` и отпирает ее вызовом функции-члена `m.unlock()`.

Синхронизация с помощью объекта `mutexCout` класса `std::mutex`

```
// mutex.cpp
...
#include <mutex>
#include <thread>

...

using namespace std;
```

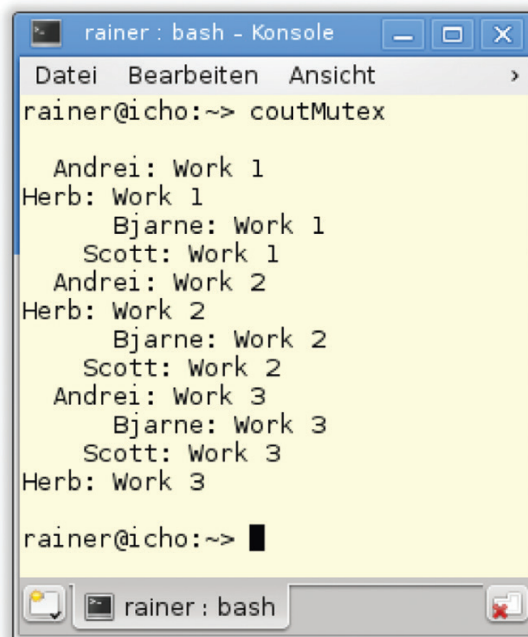
```
std::mutex mutexCout;

struct Worker{
    Worker(string n):name(n){};
    void operator() (){
        for (int i= 1; i <= 3; ++i){
            this_thread::sleep_for(chrono::milliseconds(200));
            mutexCout.lock();
            cout << name << ": " << "Work " << i << endl;
            mutexCout.unlock();
        }
    }
};

private:
    string name;
};

thread herb= thread(Worker("Herb"));
thread andrei= thread(Worker(" Andrei"));
thread scott= thread(Worker ("    Scott"));
thread bjarne= thread(Worker("    Bjarne"));
```

Каждый поток инструкций друг за другом пишет скоординированно в поток вывода данных `std::cout`, потому что используется один и тот же объект-мьютекс `mutexCout`.



```
rainer : bash - Konsole
Datei Bearbeiten Ansicht
rainer@icho:~> coutMutex

Andrei: Work 1
Herb: Work 1
    Bjarne: Work 1
    Scott: Work 1
Andrei: Work 2
Herb: Work 2
    Bjarne: Work 2
    Scott: Work 2
Andrei: Work 3
    Bjarne: Work 3
    Scott: Work 3
Herb: Work 3

rainer@icho:~> █
```

В языке C++ есть пять разных мьютексов. Они могут записывать рекурсивно, предварительно с ограничениями по времени и без ограничений по времени.

Таблица. Вариации мьютексов

Функция-член	mutex	recursive_mutex	timed_mutex	recursive_timed_mutex	shared_timed_mutex
m.lock	да	да	да	да	да
m.unlock	да	да	да	да	да
m.try_lock	да	да	да	да	да
m.try_lock_for			да	да	да
m.try_lock_until			да	да	да

Класс `std::shared_time_mutex` (коллективный хронометрированный мьютекс) позволяет реализовывать блокировочные замки на читающий и пишущий потоки инструкций. Функция-член `m.try_lock_for(relTime)` нуждается в относительной продолжительности времени; функция-член `m.try_lock_until(absTime)` – в абсолютной продолжительности времени.

Взаимная блокировка

Взаимная блокировка (неразбиваемый замок) – это состояние, при котором два или более потока инструкций заблокированы, поскольку каждый поток инструкций ожидает высвобождения ресурса другим потоком инструкций, прежде чем высвободить свой ресурс.

Взаимная блокировка может быстро возникать в ситуациях, когда забывают вызывать функцию-член `m.unlock()`. Например, это происходит при возникновении исключения в функции `getVar()`.

```
m.lock();
sharedVar= getVar();
m.unlock()
```



Не рекомендуется вызывать неизвестную функцию, удерживая блокировочный замок

Если функция `getVar` попытается получить тот же самый блокировочный замок, вызвав функцию-член `m.lock()`, то возникнет взаимная блокировка, потому что эта попытка не удастся, и вызов будет заблокирован навечно.

Еще одной типичной причиной взаимной блокировки является запираение двух мьютексов в неправильном порядке.

Взаимная блокировка

```
// deadlock.cpp
...
#include <mutex>
```

```

...

struct CriticalData{
    std::mutex mut;
};

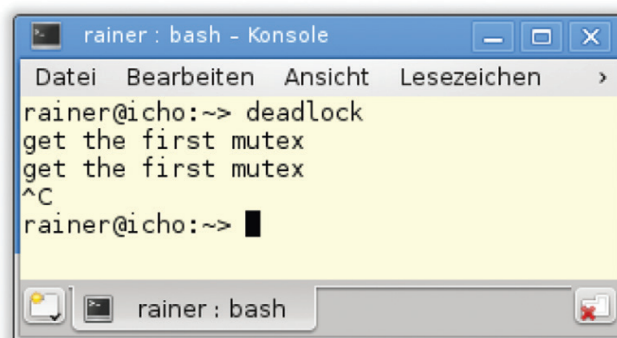
void deadLock(CriticalData& a, CriticalData& b){
    a.mut.lock();
    std::cout << "get the first mutex\n";
    std::this_thread::sleep_for(std::chrono::milliseconds(1));
    b.mut.lock();
    std::cout << "get the second mutex\n";
    a.mut.unlock(), b.mut.unlock();
}

CriticalData c1;
CriticalData c2;

std::thread t1([&]{ deadLock(c1, c2); });
std::thread t2([&]{ deadLock(c2, c1); });

```

Для того чтобы с высокой вероятностью возникла взаимная блокировка, достаточно короткого временного окна в одну миллисекунду (`std::this_thread::sleep_for(std::chrono::milliseconds(1))`), поскольку каждый поток инструкций находится вечно в ожидании другого мьютекса. В результате все становятся обездвиженными.



Рекомендуется инкапсулировать мьютекс в блокировочный замок

Очень легко забыть отпереть мьютекс или запереть мьютексы в другом порядке. В целях преодоления большинства проблем с мьютексом рекомендуется инкапсулировать его в блокировочный замок.

Блокировочные замки

Мьютекс необходимо инкапсулировать в блокировочный замок, чтобы высвобождение мьютекса выполнялось автоматически. Блокировочный замок реализован в рамках идиомы RAII, поскольку он связывает время жизни мьютекса со временем жизни блокировочного замка. В стандарте C++11 вспомогательные классы `std::lock_guard` (хранитель замка) и `std::unique_lock` (уникальный замок) предназначены соответственно для простого и для продвинутого вариантов использования. Для обоих необходим заголовок `<mutex>`. В стандарте C++14 появился вспомогательный класс `std::shared_lock` (коллективный замок), который в сочетании со вспомогательным классом `std::shared_time_mutex` (коллективным хронометрированным мьютексом) является основой для замков на читающие и пишущие потоки инструкций.

`std::lock_guard`

Вспомогательный класс `std::lock_guard` (хранитель замка) поддерживает только простой вариант использования, поэтому он может лишь связывать свой мьютекс в конструкторе и высвобождать его в деструкторе. Таким образом, синхронизация приводившегося ранее примера с работником сводится к вызову конструктора.

Синхронизация с помощью объекта класса `std::lock_guard`

```
// lockGuard.cpp
...

std::mutex coutMutex;

struct Worker{
    Worker(std::string n):name(n){};
    void operator() (){
        for (int i= 1; i <= 3; ++i){
            std::this_thread::sleep_for(std::chrono::milliseconds(200));
            std::lock_guard<std::mutex> myLock(coutMutex);
            std::cout << name << ": " << "Work " << i << '\n';
        }
    }
private:
    std::string name;
};
```

`std::unique_lock`

Использование вспомогательного класса `std::unique_lock` (уникальный замок) обходится дороже, чем использование вспомогательного класса `std::lock_guard`. С другой стороны, экземпляр класса `std::unique_lock` может создаваться

как с мьютексом, так и без него, явно записать или высвободить свой мьютекс либо откладывать записание своего мьютекса.

В следующей ниже таблице приведены функции-члены объекта `lk` вспомогательного класса `std::unique_lock`.

Таблица. Интерфейс вспомогательного класса `std::unique_lock`

Функция-член	Описание
<code>k.lock()</code>	Запирает ассоциированный мьютекс
<code>std::lock(lk1, lk2, ...)</code>	Атомарно запирает произвольное количество ассоциированных мьютексов
<code>lk.try_lock()</code> и <code>lk.try_lock_for(relTime)</code> и <code>lk.try_lock_until(absTime)</code>	Пытается запереть ассоциированный мьютекс
<code>lk.release()</code>	Высвобождает мьютекс. Мьютекс остается запертым
<code>lk.swap(lk2)</code> и <code>std::swap(lk, lk2)</code>	Меняет блокировочные замки местами
<code>lk.mutex()</code>	Возвращает указатель на ассоциированный мьютекс
<code>lk.owns_lock()</code>	Проверяет наличие мьютекса у блокировочного замка

Взаимные блокировки (неразбиваемые замки), порождаемые приобретением блокировочных замков в другом порядке, легко решаются с помощью вспомогательного класса `std::unique_lock`.

`std::unique_lock`

```
// deadLockResolved.cpp
...
#include <mutex>
...

using namespace std;

struct CriticalData{
    mutex mut;
};

void deadLockResolved(CriticalData& a, CriticalData& b){
    unique_lock<mutex>guard1(a.mut, defer_lock);
    cout << this_thread::get_id() << " : get the first lock" << endl;
    this_thread::sleep_for(chrono::milliseconds(1));
    unique_lock<mutex>guard2(b.mut, defer_lock);
```

```

    cout << this_thread::get_id() << ": get the second lock" << endl;
    cout << this_thread::get_id() << ": atomic locking";
    lock(guard1, guard2);
}

```

```

CriticalData c1;
CriticalData c2;

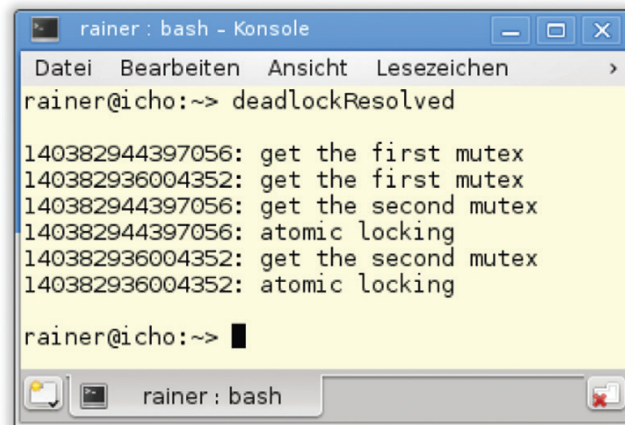
```

```

thread t1([&]{ deadLockResolved(c1, c2); });
thread t2([&]{ deadLockResolved(c2, c1); });

```

Из-за аргумента `std::defer_lock` вспомогательного класса `std::unique_lock` записание мьютексов `a.mut` и `b.mut` отлаживается. Записание происходит атомарно в вызове вспомогательной функции `std::lock(guard1, guard2)`.



std::shared_lock

Вспомогательный класс `std::shared_lock` (коллективный замок) имеет тот же интерфейс, что и вспомогательный класс `std::unique_lock`. Кроме того, вспомогательный класс `std::shared_lock` поддерживает случай, когда несколько потоков инструкций делят между собой один и тот же запертый мьютекс. В этом особом случае коллективный замок `std::shared_lock` необходимо использовать в сочетании с коллективным хронометрированным мьютексом `std::shared_timed_mutex`. Однако если один и тот же коллективный хронометрированный мьютекс `std::shared_time_mutex` используется в уникальном замке `std::unique_lock` несколькими потоками инструкций, то только один поток инструкций может им владеть.

```

#include <mutex>
...

```

```

std::shared_timed_mutex sharedMutex;
std::unique_lock<std::shared_timed_mutex> writerLock(sharedMutex);

```

```
std::shared_lock<std::shared_time_mutex> readerLock(sharedMutex);
std::shared_lock<std::shared_time_mutex> readerLock2(sharedMutex);
```

В приведенном выше примере представлен типичный сценарий записи и чтения потоков инструкций. Объект `writerLock` конкретного класса `std::unique_lock<std::shared_time_mutex>` может владеть коллективным мьютексом `sharedMutex` только исключительно, а объекты `readerLock` и `readerLock2` конкретного класса `std::shared_lock<std::shared_time_mutex>` могут владеть одинаковым коллективным мьютексом `sharedMutex` коллективно.

Потокобезопасная инициализация

Если данные не будут изменяться, то достаточно инициализировать их потокобезопасным способом. В языке C++ предлагается несколько способов достижения потоковой безопасности: использование константного выражения, использование статических переменных с блочной областью видимости либо применение утилитной функции `std::call_once` в сочетании со вспомогательным классом `std::once::flag`.

Константные выражения

Константное выражение инициализируется во время компиляции, поэтому они сами по себе являются потокобезопасными. Если перед переменной применить ключевое слово `constexpr`, то переменная становится константным выражением. Экземпляры пользовательского типа тоже могут быть константными выражениями и, следовательно, инициализироваться потокобезопасным способом, если функции-члены пользовательского типа объявлены как константные выражения.

```
struct MyDouble{
    constexpr MyDouble(double v):val(v){};
    constexpr double getValue(){ return val; }
private:
    double val
};

constexpr MyDouble myDouble(10.5);
std::cout << myDouble.getValue();           // 10.5
```

Статические переменные внутри блока

Если статическая переменная определяется в блоке, то среда исполнения C++ гарантирует, что такая переменная будет инициализирована потокобезопасным способом.

```
void blockScope(){
    static int MySharedDataInt= 2011;
}
```

std::call_once и std::once_flag

Вспомогательная функция `std::call_once` принимает два аргумента: объект вспомогательного класса `std::once_flag` и вызываемую единицу кода. С помощью объекта `std::once_flag` среда исполнения C++ гарантирует, что вызываемая единица кода будет исполнена ровно один раз.

Потокобезопасная инициализация

```
// callOnce.cpp
...
#include <mutex>

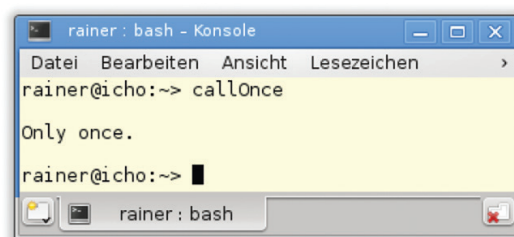
...

using namespace std;

once_flag onceFlag;
void do_once(){
    call_once(onceFlag, []{ cout << "Only once." << endl; });
}

thread t1(do_once);
thread t2(do_once);
```

Хотя оба потока инструкций исполнили функцию `do_once`, только одна из них была успешной, и лямбда-функция `[]{cout << "Only once." << endl;}` была исполнена ровно один раз.



Одинаковый объект `std::once_flag` можно использовать для регистрации разных вызываемых единиц кода, и только одна из них будет вызываться.

Потоково-локальные данные

Спецификатор хранения `thread_local` обозначает наличие потоково-локальных данных, также именуемых потоково-локальным хранилищем. У каждого потока инструкций имеется своя копия этих данных. Потоково-локальные данные ведут себя как статические переменные. Они создаются при первом использовании, и их время жизни привязано ко времени жизни потока инструкций.

Потоково-локальные данные

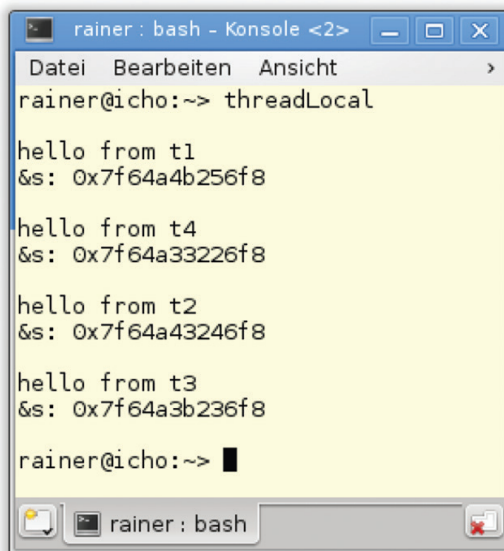
```
// threadLocal.cpp
...
std::mutex coutMutex;

thread_local std::string s("hello from ");

void addThreadLocal(std::string const& s2){
    s+= s2;
    std::lock_guard<std::mutex> guard(coutMutex);
    std::cout << s << '\n';
    std::cout << "&s: " << &s << '\n';
    std::cout << '\n';
}

std::thread t1(addThreadLocal, "t1");
std::thread t2(addThreadLocal, "t2");
std::thread t3(addThreadLocal, "t3");
std::thread t4(addThreadLocal, "t4");
```

У каждого потока инструкций имеется своя копия потоково-локального (thread_local) строкового объекта, поэтому каждый строковый объект s модифицирует свой строковый литерал независимо, и у каждого строкового литерала имеется свой уникальный адрес:



```
rainer : bash - Konsole <2>
Datei Bearbeiten Ansicht
rainer@icho:~> threadLocal

hello from t1
&s: 0x7f64a4b256f8

hello from t4
&s: 0x7f64a33226f8

hello from t2
&s: 0x7f64a43246f8

hello from t3
&s: 0x7f64a3b236f8

rainer@icho:~> █
```

Условные переменные

Условные переменные как экземпляры вспомогательного класса `std::condition_variable` позволяют синхронизировать потоки инструкций посредством сообщений. Для них необходим заголовок `<condition_variable>`. Один поток инструк-

ций выступает в роли отправителя сообщения, а другой – в роли получателя. Получатель ожидает уведомления от отправителя. Типичными вариантами использования условных переменных являются рабочие процессы производитель–потребитель.

Условная переменная может быть отправителем и получателем сообщения.

Таблица. Функции-члены объекта `cv` вспомогательного класса `std::condition_variable`

Функция-член	Описание
<code>cv.notify_one()</code>	Уведомляет ожидающий поток
<code>cv.notify_all()</code>	Уведомляет все ожидающие потоки
<code>cv.wait(lock, ...)</code>	Ожидает уведомления, удерживая при этом уникальный замок <code>std::unique_lock</code>
<code>cv.wait_for(lock, relTime, ...)</code>	Ожидает уведомления в течение определенного времени, удерживая при этом уникальный замок <code>std::unique_lock</code>
<code>cv.wait_until(lock, absTime, ...)</code>	Ожидает уведомления до тех пор, пока не наступит определенное время, удерживая при этом уникальный замок <code>std::unique_lock</code>

Отправителю и получателю нужен блокировочный замок. В случае отправителя достаточно иметь хранителя замка `std::lock_guard`, поскольку он вызывает функции-члены `lock` и `unlock` всего один раз. В случае получателя необходимо иметь уникальный замок `std::unique_lock`, поскольку он обычно запирает и отпускает свой мьютекс несколько раз.

Условная переменная

```
// conditionVariable.cpp
...
#include <condition_variable>
...

std::mutex mutex_;
std::condition_variable condVar;
bool dataReady= false;

void doTheWork(){
    std::cout << "Processing shared data." << '\n';
}

void waitingForWork(){
    std::cout << "Worker: Waiting for work." << '\n';
    std::unique_lock<std::mutex> lck(mutex_);
```

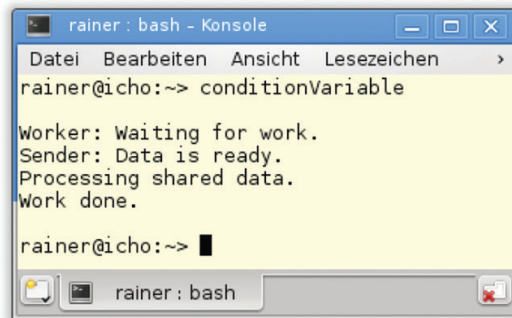
```

    condVar.wait(lck, []{ return dataReady; });
    doTheWork();
    std::cout << "Work done." << '\n';
}

void setDataReady(){
    std::lock_guard<std::mutex> lck(mutex_);
    dataReady=true;
    std::cout << "Sender: Data is ready." << '\n';
    condVar.notify_one();
}

std::thread t1(waitingForWork);
std::thread t2(setDataReady);

```



Может создаться впечатление, что использовать условную переменную просто, однако есть две важные проблемы.



Защита от мнимого пробуждения

Для того чтобы защититься от мнимого пробуждения, в вызове функции-члена `wait` условной переменной должен использоваться дополнительный предикат. Этот предикат гарантирует, что уведомление действительно пришло от отправителя. В качестве предиката можно использовать лямбда-функцию `[]{ return dataReady; }`. Переменная `dataReady` устанавливается равной `true` отправителем.



Защита от утеряннного пробуждения

Для того чтобы защититься от утерянного пробуждения, в вызове функции `wait` условной переменной должен использоваться дополнительный предикат. Этот предикат гарантирует, что уведомление отправителя не будет утеряно. Уведомление будет утеряно, если отправитель уведомит получателя до того, как тот начнет ждать, вследствие чего получатель будет ждать вечно. В связи с этим теперь получатель сначала проверяет свой предикат: `[]{ return dataReady; }`.

Семафоры

Семафоры – это механизм синхронизации, который контролирует конкурентный доступ к коллективному ресурсу. Считающий семафор – это специальный семафор со счетчиком больше нуля. Счетчик инициализируется в конструкторе. Приобретение семафора уменьшает счетчик, а высвобождение семафора увеличивает его. Если поток инструкций пытается приобрести семафор, когда счетчик равен нулю, то данный поток блокируется до тех пор, пока другой поток не увеличит счетчик, высвободив семафор.

В стандарте C++20 поддерживается класс двоичного семафора `std::binary_semaphore`, который является псевдонимом семафора с ограничением на количество разрешений `std::counting_semaphore<1>`. В этом случае наименьшее максимальное значение равно 1.

```
using binary_semaphore = std::counting_semaphore<1>;
```

В отличие от мьютекса `std::mutex`, считающий семафор `std::counting_semaphore` не привязан к потоку инструкций, в силу чего вызов функций приобретения и высвобождения может происходить в разных потоках инструкций. В следующей ниже таблице представлен интерфейс класса `std::counting_semaphore`.

Таблица. Функции-члены объекта `sem` класса `std::counting_semaphore`

Функция-член	Описание
<code>counting_semaphore::max()</code>	Возвращает максимальное значение счетчика
<code>sem.release(upd = 1)</code>	Атомарно увеличивает счетчик на величину <code>upd</code>
<code>sem.acquire()</code>	Выполняет функцию-член <code>sem.try_acquire</code> и блокирует до тех пор, пока счетчик не станет больше нуля
<code>sem.try_acquire()</code>	Атомарно уменьшает счетчик
<code>sem.try_acquire_for(relTime)</code>	Выполняет функцию-член <code>sem.try_acquire</code> в течение определенного времени
<code>sem.try_acquire_until(absTime)</code>	Выполняет функцию-член <code>sem.try_acquire</code> до тех пор, пока не наступит определенный момент времени

`std::counting_semaphore`

```
// threadSynchronisationSemaphore.cpp
#include <semaphore>
...

std::counting_semaphore<1> prepareSignal(0); // (1)

void prepareWork() {
    myVec.insert(myVec.end(), {0, 1, 0, 3});
    std::cout << "Sender: Data prepared." << '\n';
    prepareSignal.release(); // (2)
```

```

}

void completeWork() {
    std::cout << "Waiter: Waiting for data." << '\n';
    prepareSignal.acquire(); // (3)
    myVec[2] = 2;
    std::cout << "Waiter: Complete the work." << '\n';
    for (auto i: myVec) std::cout << i << " ";
    std::cout << '\n';
}

...

std::thread t1(prepareWork);
std::thread t2(completeWork);

t1.join();
t2.join();

```

Объект `prepareSignal` класса `std::counting_semaphore` (строка 1) может иметь значения 0 и 1. В данном конкретном примере он инициализирован значением 0, в силу чего вызов функции-члена `prepareSignal.release()` устанавливает значение 1 (строка 2) и разблокирует вызов функции-члена `prepareSignal.acquire()` (строка 3).

```

C:\Users\seminar>threadSynchronisationSemaphore.exe

Sender: Data prepared.
Waiter: Waiting for data.
Waiter: Complete the work.
0 1 2 3

C:\Users\seminar>threadSynchronisationSemaphore.exe

Sender: Data prepared.
Waiter: Waiting for data.
Waiter: Complete the work.
0 1 2 3

C:\Users\seminar>threadSynchronisationSemaphore.exe

Waiter: Waiting for data.
Sender: Data prepared.
Waiter: Complete the work.
0 1 2 3

C:\Users\seminar>threadSynchronisationSemaphore.exe

Sender: Data prepared.
Waiter: Waiting for data.
Waiter: Complete the work.
0 1 2 3

C:\Users\seminar>

```

Координационные типы

Защелки и барьеры – это координационные типы, которые позволяют некоторым потокам инструкций блокировать до тех пор, пока счетчик не станет равным нулю. В стандарте C++20 защелки и барьеры реализованы в двух вариациях: во вспомогательном классе `std::latch` и во вспомогательном классе `std::barrier`.

`std::latch`

Теперь давайте подробнее рассмотрим интерфейс вспомогательного класса `std::latch`.

Таблица. Функции-члены объекта `lat` вспомогательного класса `std::latch`

Функция-член	Описание
<code>lat.count_down(upd = 1)</code>	Атомарно уменьшает счетчик на величину <code>upd</code> без блокирования вызывающей стороны
<code>lat.try_wait()</code>	Возвращает <code>true</code> , если счетчик равен 0
<code>lat.wait()</code>	Возвращает немедленно, если счетчик равен 0. Если нет, то блокирует до тех пор, пока счетчик не станет равным 0
<code>lat.arrive_and_wait(upd = 1)</code>	Эквивалентно вызовам <code>count_down(upd); wait();</code>

Дефолтное значение аргумента `upd` равно 1. Если величина `upd` больше счетчика либо является отрицательной, то программа ведет себя неопределенно. Из названия может показаться, что при вызове функции-члена `lat.try_wait()` она начинает ждать. Однако это не так. Она никогда не ждет. Вместо этого она проверяет состояние защелки и сразу возвращает значение.

В следующей ниже программе используется два объекта-защелки `std::latch`, чтобы построить рабочий процесс начальник–работники. Запись в поток вывода данных `std::cout` синхронизируется с помощью функции `synchronizedOut` (строка 13). Эта синхронизация облегчает отслеживание рабочего процесса.

`std::latch`

```
// bossWorkers.cpp
#include <latch>

...

std::latch workDone(2);
std::latch goHome(1);           // (5)

void synchronizedOut(const std::string s) {
    std::lock_guard<std::mutex> lo(coutMutex);
    std::cout << s;
```

```

}

class Worker {
public:
    Worker(std::string n): name(n) { };

    void operator() (){
        // уведомить начальника о завершении работы
        synchronizedOut(name + ": " + "Work done!\n");
        workDone.count_down();           // (3)

        // ожидание перед отправкой домой
        goHome.wait();
        synchronizedOut(name + ": " + "Good bye!\n");
    }
private:
    std::string name;
};

...

std::cout << "BOSS: START WORKING! " << '\n';

Worker herb("  Herb");                // (1)
std::thread herbWork(herb);

Worker scott("  Scott");                // (2)
std::thread scottWork(scott);

workDone.wait();                       // (4)

std::cout << '\n';

goHome.count_down();

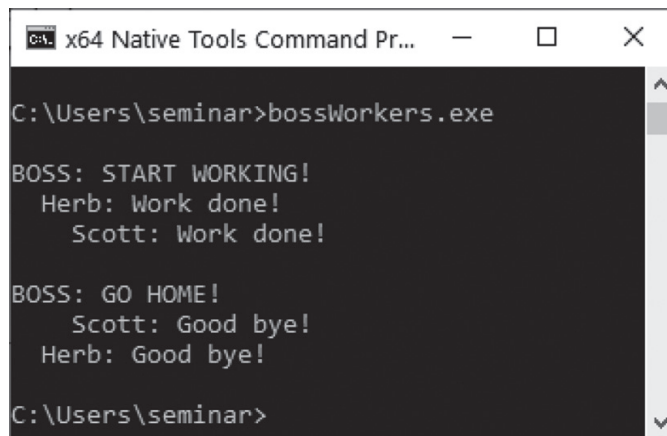
std::cout << "BOSS: GO HOME!" << '\n';

herbWork.join();
scottWork.join();

```

Идея приведенного выше рабочего процесса проста. Два работника, Херб и Скотт (строки 1 и 2), должны выполнить свою работу. По окончании своей работы (строка 3) они вычитают единицу из объекта-защелки `std::latch workDone`. Босс (главный поток инструкций) блокируется в строке (4) до тех пор, пока счетчик не станет равным 0. Когда счетчик становится равным 0, босс использует второй объект-защелку `std::latch goHome`, чтобы дать сигнал сво-

им работникам отправляться домой. В этом случае первоначальный счетчик равен 1 (строка 5). Вызов функции-члена `goHome.wait(0)` блокирует до тех пор, пока счетчик не станет равным 0.



```

C:\Users\seminar>bossWorkers.exe

BOSS: START WORKING!
  Herb: Work done!
  Scott: Work done!

BOSS: GO HOME!
  Scott: Good bye!
  Herb: Good bye!

C:\Users\seminar>

```

std::barrier

Вспомогательный класс `std::barrier` похож на вспомогательный класс `std::latch`. Однако между обоими классами есть два различия. Во-первых, барьер `std::barrier` можно использовать более одного раза, а во-вторых, можно устанавливать счетчик для следующего шага (итерации). Сразу после того, как счетчик становится равным нулю, начинается так называемый шаг завершения. На этом шаге вызывается вызываемая единица кода¹. Барьер `std::barrier` получает свою вызываемую единицу кода в своем конструкторе.

На шаге завершения выполняются следующие ниже действия:

1. Все потоки инструкций блокируются.
2. Произвольный поток разблокируется и исполняет вызываемую единицу кода.
3. Если шаг завершения закончен, то все потоки разблокируются.

Таблица. Функции-члены вспомогательного класса `std::barrier`

Функция-член	Описание
<code>bar.arrive(upd)</code>	Атомарно уменьшает счетчик на величину <code>upd</code>
<code>bar.wait()</code>	Блокирует в точке синхронизации до тех пор, пока не будет выполнен шаг завершения
<code>bar.arrive_and_wait()</code>	Эквивалентно вызову <code>wait(arrive())</code>
<code>bar.arrive_and_drop()</code>	Уменьшает счетчик для текущей и последующей фаз на единицу
<code>std::barrier::max</code>	Максимальное значение, которое поддерживается реализацией

¹ См. https://en.cppreference.com/w/cpp/named_req/Callable.

Задания

Дополнительно к потокам инструкций в языке C++ есть задания, которые служат для того, чтобы выполнять работу асинхронно. Заданиям необходим заголовок `<future>`. Задание параметризуется рабочим пакетом, состоящим из двух ассоциированных компонентов: объекта-обещания и объекта-будущего. Оба компонента связаны между собой каналом данных. Обещание выполняет рабочие пакеты и помещает результат в канал данных; связанное с ним будущее подхватывает результат. Обе конечные точки связи могут выполняться в отдельных потоках инструкций. Особенностью является то, что объект-будущее может забирать результат позже, и поэтому вычисление результата объектом-обещанием не зависит от запроса результата ассоциированным с ним объектом-будущим.

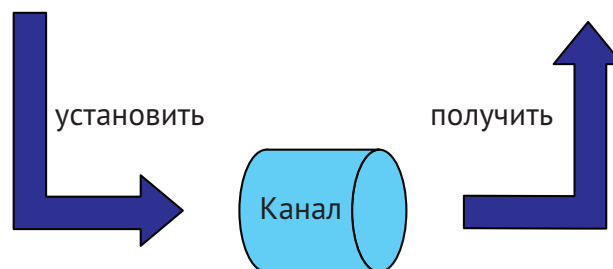


К заданиям следует относиться как к каналам данных

Задания ведут себя как каналы данных. Обещание помещает свой результат в канал данных. Будущее ожидает результат и его подхватывает.

Обещание: отправитель

Будущее: получатель



Потоки инструкций в сопоставлении с заданиями

Потоки инструкций кардинально отличаются от заданий. Для связи между потоком-создателем и создаваемым потоком инструкций необходимо использовать коллективную переменную. Задание общается через свой канал данных, который неявно защищен, и поэтому задание не должно использовать механизм защиты наподобие мьютекса.

Поток-создатель ожидает свой дочерний поток инструкций посредством вызова функции-члена `join`. Объект-будущее `fut` использует вызов функции-члена `fut.get()`, который блокирует, если результата нет.

Если в созданном потоке инструкций происходит исключение, то созданный поток инструкций, создатель и весь процесс завершатся. Напротив, обещание может посылать исключения в будущее, которое должно обрабатывать эти исключения.

Обещание может обслуживать одно или много объектов-будущих. Оно может отправлять значение, исключение либо только уведомление. Задание можно использовать как безопасную замену условной переменной.

```

#include <future>
#include <thread>

...

int res;
std::thread t([&]{ res= 2000+11;});
t.join();
std::cout << res << '\n';           // 2011

auto fut= std::async([]{ return 2000+11; });
std::cout << fut.get() << '\n';     // 2011

```

Дочерний поток инструкций `t` и вызов вспомогательной функции `std::async` запуска асинхронных заданий вычисляет сумму 2000 и 11. Поток-создатель получает результат от своего дочернего потока инструкций `t` через коллективную переменную `res`. Вызов вспомогательной функции `std::async` создает канал данных между отправителем (обещанием) и получателем (будущим). Будущее запрашивает канал данных с помощью функции-члена `fut.get()`, чтобы получить результат вычисления. Вызов функции-члена `fut.get()` является блокирующим.

std::async

Вспомогательная функция высшего порядка `std::async` ведет себя как вызов асинхронной функции. Этот вызов принимает вызываемую единицу кода и ее аргументы. Функция `std::async` является вариативным шаблоном и поэтому может принимать произвольное количество аргументов. Вызов функции `std::async` возвращает объект-будущее `fut`. Это манипулятор для получения результата посредством функции-члена `fut.get()`. Для вспомогательной функции `std::async` опционально можно указывать политику запуска. С помощью политики запуска можно явно определять, должна ли асинхронная операция выполняться в том же потоке инструкций (`std::launch::deferred`) либо в другом потоке (`std::launch::async`).

Вызов `auto fut= std::async(std::launch::deferred, ...)` не будет исполнен немедленно. Вызов функции-члена `fut.get()` запускает обещание лениво.

Режимы ленивого и немедленного исполнения вспомогательной функцией `std::async`

```

// asyncLazyEager.cpp
...
#include <future>

...

using std::chrono::duration;
using std::chrono::system_clock;

```

```

using std::launch;

auto begin= system_clock::now();

auto asyncLazy= std::async(launch::deferred, []{ return system_clock::now(); });
auto asyncEager= std::async(launch::async, []{ return system_clock::now(); });
std::this_thread::sleep_for(std::chrono::seconds(1));

auto lazyStart= asyncLazy.get() - begin;
auto eagerStart= asyncEager.get() - begin;

auto lazyDuration= duration<double>(lazyStart).count();
auto eagerDuration= duration<double>(eagerStart).count();

std::cout << lazyDuration << " sec";           // 1.00018 sec.
std::cout << eagerDuration << " sec";         // 0.00015489 sec.

```

Результат работы программы показывает, что обещание, ассоциированное с объектом-будущим `asyncLazy`, выполняется на одну секунду позже, чем обещание, ассоциированное с объектом-будущим `asyncEager`. Одна секунда – это как раз то время, в течение которого создатель спит, после чего объект-будущее `asyncLazy` запрашивает свой результат.



При выборе вспомогательная функция `std::async` должна стоять на первом месте

Среда исполнения C++ определяет, будет функция `std::async` выполняться в отдельном потоке инструкций или нет. Решение среды исполнения C++ может зависеть от количества ядер, загрузки системы или размера рабочего пакета.

`std::packaged_task`

Вспомогательный класс `std::packaged_task` (упакованное задание) представляет собой обертку вокруг вызываемой единицы кода, которая может быть исполнена асинхронно. Данная обертка впоследствии может быть исполнена в отдельном потоке инструкций.

Поэтому необходимо выполнить четыре шага.

1. Обернуть свою работу:

```
std::packaged_task<int(int, int)> sumTask([](int a, int b){ return a+b; });
```

2. Создать объект-будущее:

```
std::future<int> sumResult= sumTask.get_future();
```

3. Выполнить вычисление:

```
sumTask(2000, 11);
```

4. Запросить результат:

```
sumResult.get();
```

Упакованное задание `std::packaged_task` либо будущее `std::future` можно перемещать в отдельный поток инструкций.

`std::packaged_task`

```
// packaged_task.cpp
...
#include <future>

...

using namespace std;

struct SumUp{
    int operator()(int beg, int end){
        for (int i= beg; i < end; ++i ) sum += i;
        return sum;
    }
private:
    int beg;
    int end;
    int sum{0};
};

SumUp sumUp1, sumUp2;

packaged_task<int(int, int)> sumTask1(sumUp1);
packaged_task<int(int, int)> sumTask2(sumUp2);

future<int> sum1= sumTask1.get_future();
future<int> sum2= sumTask2.get_future();

deque< packaged_task<int(int, int)>> allTasks;
allTasks.push_back(move(sumTask1));
allTasks.push_back(move(sumTask2));

int begin{1};
int increment{5000};
int end= begin + increment;

while (not allTasks.empty()){
```

```

    packaged_task<int(int, int)> myTask= move(allTasks.front());
    allTasks.pop_front();
    thread sumThread(move(myTask), begin, end);
    begin= end;
    end += increment;
    sumThread.detach();
}

auto sum= sum1.get() + sum2.get();
cout << sum;                // 50005000

```

Обещания (упакованное задание `std::packaged_task`) перемещаются в очередь `std::deque`. Программа выполняет итерации в цикле `while` по всем обещаниям. Каждое обещание выполняется в своем потоке инструкций и в фоновом режиме выполняет сложение (`sumThread.detach()`). Результатом является сумма всех чисел от 1 до 100000.

std::promise и std::future

Пара вспомогательных классов `std::promise` и `std::future` обеспечивает полный контроль над заданием.

Таблица. Функции-члены объекта-обещания `prom` вспомогательного класса `std::promise`

Функция-член	Описание
<code>prom.swap(prom2)</code> и <code>std::swap(prom, prom2)</code>	Меняет обещания местами
<code>prom.get_future()</code>	Возвращает объект-будущее
<code>prom.set_value(val)</code>	Устанавливает значение
<code>prom.set_exception(ex)</code>	Устанавливает исключение
<code>prom.set_value_at_thread_exit(val)</code>	Сохраняет значение и делает его готовым в случае, если обещание прекращается в момент завершения потока инструкций
<code>prom.set_exception_at_thread_exit(ex)</code>	Сохраняет исключение и делает его готовым в случае, если обещание прекращается в момент завершения потока инструкций

Если объект-обещание устанавливает значение либо исключение более одного раза, то генерируется исключение `std::future_error`.

Таблица. Функции-члены объекта-будущего `std::future`

Функция-член	Описание
<code>fut.share()</code>	Возвращает коллективное будущее <code>std::shared_future</code>
<code>fut.get()</code>	Возвращает результат, который может быть значением либо исключением
<code>fut.valid()</code>	Проверяет наличие результата. Возвращается после того, как вызов функции-члена <code>fut.get()</code> дает <code>false</code>
<code>fut.wait()</code>	Ожидает результата
<code>fut.wait_for(relTime)</code>	Ожидает результата в течение определенного времени
<code>fut.wait_until(absTime)</code>	Ожидает результата до тех пор, пока не наступит определенное время

Если объект-будущее `std::future fut` запрашивает результат более одного раза, то генерируется исключение `std::future_error`. Объект-будущее создает коллективное будущее путем вызова функции-члена `fut.share()`. Такого рода объекты связаны со своим обещанием и могут запрашивать результат независимо. Коллективное будущее имеет тот же интерфейс, что и будущее.

Ниже приведен пример использования объекта-обещания и объекта-будущего.

Объект-обещание и объект-будущее

```
// promiseFuture.cpp
...
#include <future>
...

void product(std::promise<int>&& intPromise, int a, int b){
    intPromise.set_value(a*b);
}

int a= 20;
int b= 10;

std::promise<int> prodPromise;
std::future<int> prodResult= prodPromise.get_future();

std::thread prodThread(product, std::move(prodPromise), a, b);
std::cout << "20*10= " << prodResult.get();           // 20*10= 200
```

Объект-обещание `prodPromise` перемещается в отдельный поток инструкций и выполняет свое вычисление. Объект-будущее получает результат с помощью функции-члена `prodResult.get()`.

Синхронизация

Объект-будущее `fut` может быть синхронизировано со связанным с ним объектом-обещанием вызовом функции-члена `fut.wait()`. В отличие от условных переменных, блокировочные замки и мьютексы не требуются, а мнимые и утерянные пробуждения невозможны.

Задания для синхронизации

```
// promiseFutureSynchronise.cpp
...
#include <future>

...

void doTheWork(){
    std::cout << "Processing shared data." << '\n';
}

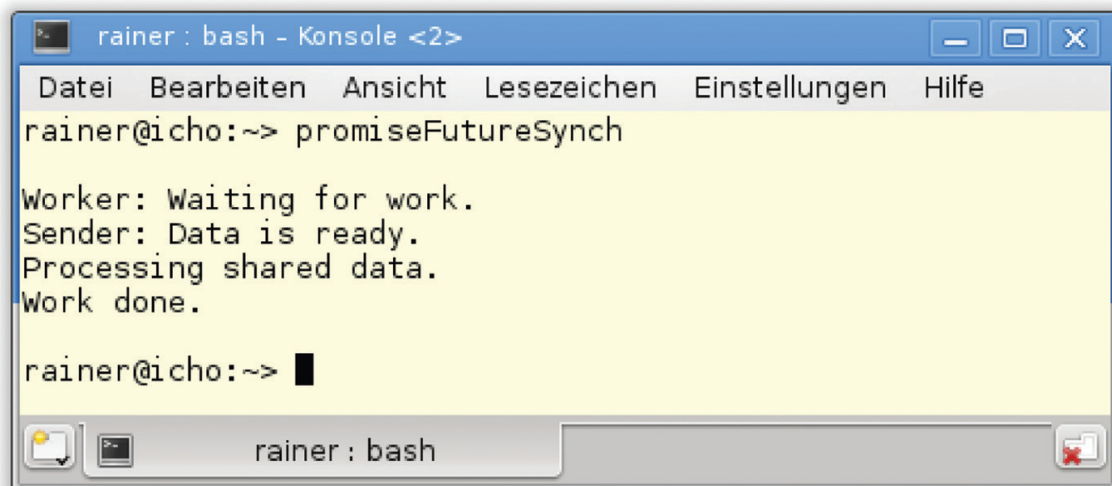
void waitingForWork(std::future<void>&& fut){
    std::cout << "Worker: Waiting for work." << '\n';
    fut.wait();
    doTheWork();
    std::cout << "Work done." << '\n';
}

void setDataReady(std::promise<void>&& prom){
    std::cout << "Sender: Data is ready." << '\n';
    prom.set_value();
}

std::promise<void> sendReady;
auto fut= sendReady.get_future();

std::thread t1(waitingForWork, std::move(fut));
std::thread t2(setDataReady, std::move(sendReady));
```

Вызов функции-члена `prom.set_value()` на объекте-обещании пробуждает объект-будущее, которое затем может выполнить свою работу.

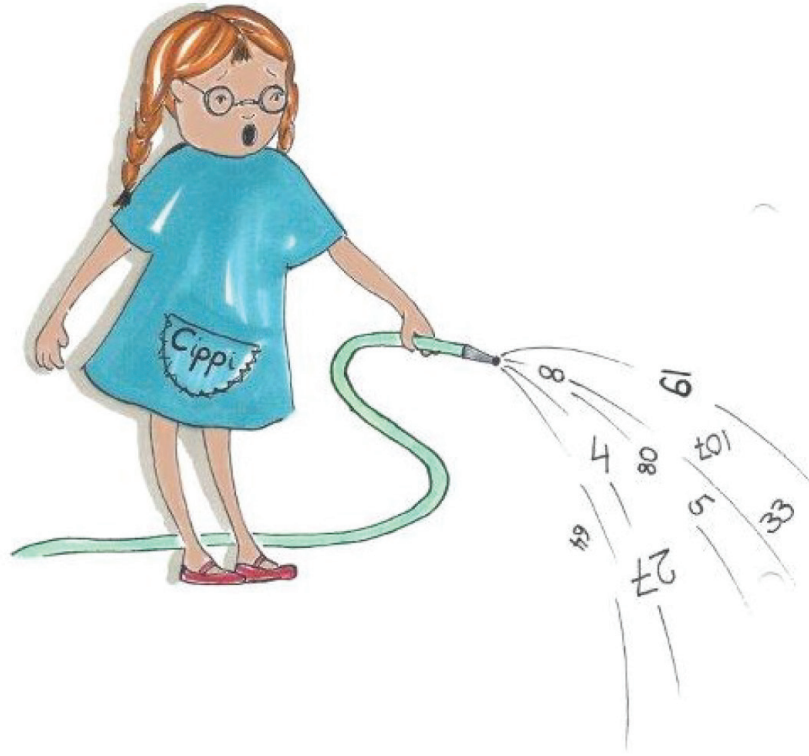


```
rainer : bash - Konsole <2>
Datei Bearbeiten Ansicht Lesezeichen Einstellungen Hilfe
rainer@icho:~> promiseFutureSynch

Worker: Waiting for work.
Sender: Data is ready.
Processing shared data.
Work done.

rainer@icho:~> █
```

20. Сопрограммы



Циппи поливает цветы

В стандарте C++20 конкретных сопрограмм не предлагалось. Однако в этом стандарте имелся фреймворк для реализации сопрограмм. С другой стороны, в стандарте C++23 благодаря генератору сопрограмм `std::generator` в библиотеке `ranges` появилась первая конкретная сопрограмма. Настоящая глава дает лишь приблизительное представление о сложном фреймворке сопрограмм.

Краткий обзор

Сопрограммы (корутины) – это функции, которые могут приостанавливать и возобновлять свое исполнение, сохраняя при этом свое состояние. В связи с этим сопрограмма состоит из трех частей: объекта-обещания, дескриптора сопрограммы и фрейма сопрограммы.

- Объект-обещание управляется изнутри сопрограммы и возвращает свой результат через объект-обещание.
- Дескриптор сопрограммы – это невладеющий дескриптор для возобновления или уничтожения фрейма сопрограммы извне.
- Фрейм сопрограммы – это внутреннее состояние, обычно размещенное в куче. Он состоит из уже упомянутого объекта-обещания, копируемых параметров сопрограммы, представления точки приостановки, локаль-

ных переменных, время жизни которых заканчивается перед текущей точкой приостановки, и локальных переменных, время жизни которых превышает текущую точку приостановки.

В целях оптимизации выделяемых ресурсов под хранение состояния сопрограммы необходимо соблюдать ряд требований. Функция неявно становится сопрограммой, если в ней вместо ключевого слова `return` используется ключевое слово `co_return`, а также ключевые слова `co_yield` или `co_await`.

Новые ключевые слова расширяют возможности выполнения функций C++ двумя новыми понятиями.

co_yield

Ключевое слово `co_yield` позволяет писать функцию-генератор. Функция-генератор каждый раз возвращает новое значение. Данная функция представляет собой поток данных, из которого можно забирать значения. Поток данных может быть бесконечным.

Функция-генератор

```
Generator<int> getNext(int start = 0, int step = 1) noexcept {
    auto value = start;
    for (int i = 0;; ++i){
        co_yield value;
        value += step;
    }
}
```

co_await

Ключевое слово `co_await` позволяет приостанавливать и возобновлять исполнение следующего после него выражения. При использовании выражения с ключевым словом `co_await` в функции `func` вызов `auto getResult = func()` не блокирует, если результат функции недоступен. Вместо ресурсоемкого блокирования получается дружественное для ресурсов ожидание. Выражение должно допускать ожидание, то есть быть так называемым «ожидаемым» (awaitable) объектом, и должно поддерживать следующие три функции-члена: `await_ready`, `await_suspend` и `await_resume`.

Сопрограмма

```
Acceptor acceptor{443};

while (true){
    Socket socket= co_await acceptor.accept();
```

```

    auto request= co_await socket.read();
    auto response= handleRequest(request);
    co_await socket.write(response);
}

```

«Ожидаемое» выражение `expr` после ключевого слова `co_await` должно реализовывать функции-члены `await_ready`, `await_suspend` и `await_resume`.

Типы, допускающие ожидание

В стандарте C++20 уже определены два «ожидаемых» вспомогательных класса в качестве базовых строительных блоков в контексте сопрограмм: `std::suspend_always` и `std::suspend_never`.

Предопределенные «ожидаемые» вспомогательные классы

```

struct suspend_always {
    constexpr bool await_ready() const noexcept { return false; }
    constexpr void await_suspend(coroutine_handle<>) const noexcept {}
    constexpr void await_resume() const noexcept {}
};

struct suspend_never {
    constexpr bool await_ready() const noexcept { return true; }
    constexpr void await_suspend(coroutine_handle<>) const noexcept {}
    constexpr void await_resume() const noexcept {}
};

```

Бесконечный поток данных посредством `co_yield`

Следующая ниже программа создает бесконечный поток данных. В сопрограмме `getNext` используется ключевое слово `co_yield`, чтобы создать поток данных, который начинается со `start` и выдает по запросу следующее значение, увеличенное на шаг `step`.

Бесконечный поток данных

```

// infiniteDataStream.cpp
...
#include <coroutine>

template<typename T>
struct Generator {

```

```

struct promise_type;
using handle_type = std::coroutine_handle<promise_type>;

Generator(handle_type h): coro(h) {} // (3)
handle_type coro;

~Generator() {
    if ( coro ) coro.destroy();
}
Generator(const Generator&) = delete;
Generator& operator = (const Generator&) = delete;
Generator(Generator&& oth) noexcept : coro(oth.coro) {
    oth.coro = nullptr;
}

Generator& operator = (Generator&& oth) noexcept {
    coro = oth.coro;
    oth.coro = nullptr;
    return *this;
}
T getValue() {
    return coro.promise().current_value;
}
bool next() { // (5)
    coro.resume();
    return not coro.done();
}
struct promise_type {
    promise_type() = default; // (1)

    ~promise_type() = default;

    auto initial_suspend() { // (4)
        return std::suspend_always{};
    }
    auto final_suspend() noexcept {
        return std::suspend_always{};
    }
    auto get_return_object() { // (2)
        return Generator{handle_type::from_promise(*this)};
    }
    auto return_void() {
        return std::suspend_never{};
    }

    auto yield_value(const T value) { // (6)
        current_value = value;
    }
}

```

```

        return std::suspend_always{};
    }
    void unhandled_exception() {
        std::exit(1);
    }
    T current_value;
};

Generator<int> getNext(int start = 0, int step = 1){
    auto value = start;
    for (int i = 0;; ++i){
        co_yield value;
        value += step;
    }
}

int main() {

    std::cout << '\n';

    std::cout << "getNext():";
    auto gen = getNext();
    for (int i = 0; i <= 10; ++i) {
        gen.next();
        std::cout << " " << gen.getValue();           // (7)
    }

    std::cout << '\n';

}

```

Главная программа main создает сопрограмму. Сопрограмма gen(7) возвращает значения от 0 до 10.

```
getNext(): 0 1 2 3 4 5 6 7 8 9 10
```

Бесконечный поток данных

Числа в программе infiniteDataStream.cpp означают первую итерацию рабочего процесса. Данная программа:

- 1) создает объект-обещание;
- 2) вызывает функцию-член promise.get_return_object() и сохраняет результат в локальной переменной;
- 3) создает генератор;

- 4) вызывает функцию-член `promise.initial_suspend()`. Генератор является ленивым и поэтому всегда приостанавливается;
- 5) запрашивает следующее значение и возвращается, если генератор потреблен;
- 6) инициирует обращение к `co_yield`, после чего становится доступным следующее значение;
- 7) получает следующее значение.

В дополнительных итерациях выполняются только шаги 5 и 6.

Предметный указатель

#

#include 25

::

::operator delete 26

::operator new 26

—

_copy 142

_if 142

<

<algorithm> 29, 30, 141

<any> 68

<array> 89

<atomic> 281

<cmath> 31, 204

<condition_variable> 307

<cstdio> 26

<cstdlib> 205

<deque> 93

<expected> 75

<format> 262

<forward_list> 96

<fstream> 255

<functional> 36, 41, 138

<future> 315

<iomanip> 251

<iostream> 247, 251, 258

<istream> 247

<iterator> 131, 133

<list> 94

<map> 103

<memory> 43, 187

<mutex> 298, 302

<new> 26

<numbers> 206

<numeric> 31, 141, 181

<optional> 69

<ostream> 247

<print> 263

<queue> 114, 115

<random> 201

<regex> 231

<set> 103

<sstream> 253

<stack> 113

<stdio.h> 26

<string> 208, 217

<string_view> 226

<thread> 287

<tuple> 39

<type_traits> 29

<utility> 29, 32, 33, 35, 38

<variant> 72

<vector> 90

A

auto 130

await_ready 324

await_resume 324

await_suspend 324

B

basic_ios<> 246

basic_iostream<> 246

basic_istream<> 246

basic_ostream<> 246

basic_streambuf<> 246

bidirectional_iterator 191

BitmaskType 272

boost::filesystem 269

Boost, библиотеки 68

C

char 209, 227, 247

char16_t 209, 227

co_yield 324, 325

cstdlib 22

CTAG 123

D

default 282

E

en.cppreference.com 67

Enum 35

F

filesystem 23

format 23

Fortran 125

forward_iterator 191

G

GCC, компилятор C++ 69

getline 216

I

import std 26

ios_base 246

iostream 23

J

jthread 24

L

limits 22

M

MATLAB 125

max 29

memcmp 283

memcpy 283

Microsoft Visual C++,
компилятор C++ 69

min 29

minmax 29

N

null 212

numeric 22

P

POSIX 272

printf 23, 262

Python 125, 262, 264

R

RAII (Приобретение ресурсов
есть инициализация) 43, 302

random 22

random_access_iterator 191

ranges 22

regex 22

S

seed 166

static_assert 53

std::

equal_range 171

accumulate 181

adjacent_difference 181

advance 131

all_of 150

any 18, 19, 68

any_of 150

array 20, 21, 79, 89, 116, 119, 122

async 316

atomic 281, 303

atomic<floating-point type> 283

atomic<integral type> 284

atomic<smart T*> 283

atomic<T*> 283

atomic_flag 280

atomic_ref 285

auto_ptr 43

back_inserter 133

bad_any_cast 68

bad_expected_access 75

bad_optional_access 70

bad_variant_access 72, 73

barrier 24, 314

basic_common_reference 58

basic_string_view 227

begin 131

binary_search 1

- chrono 61, 62, 64, 65, 68, 275
- cin 134, 248
- clamp 179
- cmp_equal 32
- cmp_greater 32
- cmp_greater_equal 32
- cmp_less 32
- cmp_less_equal 32
- cmp_not_equal 32
- common_reference 58
- common_type 58
- conditional 58
- conjunction 58
- copy 155
- copy_backward 155
- copy_if 155
- counting_semaphore 310
- cout 135, 248, 296, 299
- cref 19
- default_random_engine 202
- defer_lock 304
- deque 20, 79, 88, 93, 116
- disjunction 58
- distance 131
- enable_if 58
- end 131
- endl 248
- equal 151
- equal_to<key> 107
- erase 85
- exclusive_scan 181
- expected 75
- filesystem 271, 272, 273, 274, 275
- fill 159
- fill_n 159
- find_end 153
- find_if 209
- flat_map 116
- flat_multimap 116
- flat_set 116
- for_each 146
- format 262, 263
- format_error 267
- formatter 264, 266
- format_to 262, 263
- format_to_n 262, 263
- forward 29, 33
- forward_list 20, 79, 88, 96
- front_inserter 133
- function 19, 35, 37
- future 318, 319, 320
- future_error 319, 320
- generate 159
- generate_n 159
- generator 18, 195
- get 39
- get_if 72
- gmtime 275
- greater 138
- hash<key> 107
- identity 200
- ignore 40
- includes 173
- inclusive_scan 181
- inner_product 181
- inplace_merge 173
- inserter 133
- invalid_argument 224
- ios 259, 260
- iota 181
- is_constant_evaluated 59
- is_corresponding_member 59
- is_heap 176
- is_heap_until 176
- is_partitioned 167
- is_pointer_interconvertible_with_
 - class 59
- is_sorted 169
- is_sorted_until 169
- istringstream 254
- jthread 24, 292
- latch 24, 312
- lerp 31
- less 104, 115, 138, 169, 171, 173,
 - 176, 180
- lexicographical_compare 151
- lexicographical_compare_three_way
 - 151
- list 20, 79, 88, 94, 96
- localtime 275
- lock_guard 302, 308
- lower_bound 171
- make_any.any.emplace 68
- make_format_args 262, 263
- make_heap 176
- make_optional 69
- make_pair 39
- make_shared 48
- make_tuple 40
- make_unique 47
- map 20, 38, 99, 103, 104, 106, 107,
 - 116

- match_results 234
- match_results.format 241
- max 30
- max_element 178
- mdspan 122
- merge 173
- midpoint 31
- min 30
- min_element 178
- minmax 30
- minmax_element 178
- mismatch 151
- move 29, 33, 83, 160
- move_backward 160
- mt19937 203
- multimap 20, 38, 99, 103, 116
- multiset 20, 99, 103, 116
- mutex 298, 310
- negation 58
- next 131
- next_permutation 180
- none_of 150
- nth_element 169
- number 206
- once 305
- once_flag 306
- optional 18, 19, 68, 69
- ostreamstream 254
- osyncstream 298
- out_of_range 92, 106, 215, 224
- packaged_task 317
- pair 19, 30, 38, 39
- partial_sort 169
- partial_sort_copy 169
- partial_sum 181
- partition 167
- partition_copy 167
- partition_point 167
- pop_heap 176
- prev 131
- prev_permutation 180
- print 262, 263
- printf 26
- println 262, 263
- priority_queue 21, 112, 115
- promise 319, 320
- push_heap 176
- queue 21, 112, 114
- random_access_iterator 200
- random_device 202
- random_shuffle 165
- range-error 89
- ranges 190
- ranges 191, 193
- ratio 64
- reduce 181
- ref 19, 42
- reference_wrapper 41
- regex_constants 233, 240
- regex_iterator 231, 242
- regex_match 231, 234
- regex_replace 231, 239
- regex_search 231, 234, 238
- regex_token_iterator 231, 242
- remove 157
- remove_copy 157
- remove_copy_if 157
- remove_if 157
- replace 156
- replace_copy 156
- replace_copy_if 156
- replace_if 156
- reverse 143, 162
- reverse_copy 162
- rotate 143, 163
- rotate_copy 163
- search 153
- search_n 153
- sentinel_for 200
- set 20, 99, 103, 116
- set_difference 173
- set_intersection 173
- set_multiset 116
- set_symmetric_difference 173
- set_union 173
- shared_lock 302, 304
- shared_ptr 19, 43, 47
- shared_ptr_from_this 49
- shared_timed_mutex 304
- shift_left 164
- shift_right 164
- shuffle 165
- sort 144, 169
- sortable 200
- sorted_unique 117
- sort_heap 176
- span 21, 118
- stable_partition 167, 168
- stable_sort 169
- stack 21, 112, 113
- stop_callback 293, 294
- stop_source 293, 294

stop_token 293
 string 21, 22, 86, 109, 119, 122, 146, 209, 227
 string 218
 stringstream 254
 string_view 23, 121, 227
 sub_match 234, 236
 suspend_always 325
 suspend_never 325
 swap 29, 35, 38, 39, 83, 161
 swap_ranges 161
 system_error 291
 terminate 288, 292
 thread 287, 289, 292
 thread 291
 tie 40
 to_underlying 29, 35
 transform 161
 transform_exclusive_scan 181
 transform_inclusive_scan 181
 transform_reduce 181
 tuple 19, 39, 89
 type_identity 58
 underlying_type 35
 unexpected 75
 unique 166
 unique_copy 166
 unique_lock 302
 unique_ptr 19, 43, 44
 unordered_map 20, 38, 99, 107
 unordered_multimap 20, 38, 99, 107
 unordered_multiset 20, 99, 107
 unordered_set 20, 99, 107
 upper_bound 171
 variant 19, 68, 72
 vector 20, 21, 79, 88, 90, 102, 116, 119, 122, 146, 209, 213
 vformat 262, 263
 vformat_to 263
 vformat_to 262
 views 193, 198
 visit 73
 void_t 58
 wcerr 247
 wcin 247
 wclog 247
 wcout 247
 weak_ptr 19, 43, 49
 wstringstream 254
 wostringstream 254
 wosyncstream 298

wstring 109
 wstringstream 254
 bind_front 19
 stop_token 294
 string 22

T

thread 23
 thread_local 306
 typeid 69
 type_traits 19, 35, 52, 57
 type-traits 29

U

UTC (Всемирное координированное время) 275

V

volatile 53

W

wchar_t 209, 227, 247

A

Адаптер итераторный 133
 вставки 133
 поточный 134
 Адаптер контейнера 21, 112
 Адаптер функциональных единиц
 кода 35
 Алгоритм 21, 140
 Алгоритм стандартный 209
 Аналог диапазонный 199

Б

База данных часовых поясов IANA 67
 Барьер 312
 Библиотека для работы с
 диапазонами 22
 Библиотека для работы
 со временем 60
 Библиотека для работы
 со случайными числами 22
 Библиотека для работы
 с потоками ввода-вывода 23
 Библиотека для работы
 с признаками типов 29

Библиотека для работы
с файловой системой 23, 269
Библиотека для работы
с форматированием 23, 262
Библиотека для работы с числами 22
Библиотеки для работы
с многопоточностью 23
Блокировка. См. Замок
блокировочный
Блокировка взаимная 300

В

Вид 190
Вид на строковый объект 226
Выделитель памяти 103
Выражение константное 305
Выражение регулярное 23, 231

Г

Генератор случайных чисел 201
Генератор сопрограмм 195
Гонка за данными 296

Д

Диапазон 21, 190
Диапазон полуоткрытый 143

Е

Единица кода вызываемая 21, 73, 136

З

Задание 24, 315
Замок блокировочный 23, 296, 302
Захват переменных 139
Защелка 312

И

Инициализация агрегатная 79
Инструмент вспомогательный 19
Итератор 21, 128, 140
Итератор ввода 130, 190
Итератор
впереднаправленный 129, 190
Итератор вывода 130, 190
Итератор двунаправленный 129, 190
Итератор концевой 21
Итератор начальный 21

Итератор со сплошным
доступом 129, 190
Итератор с произвольным
доступом 129, 190, 200, 209

К

Календарь 19
Конкретизация формата 265
стандартная 264
Константа математическая 22
Конструктор последовательности 91
Контейнер ассоциативный 20, 38, 99
неупорядоченный 21, 107
упорядоченный 20, 103
Контейнер
последовательности 20, 87
Концепт 190, 200
Кортеж 39
Корутина. См. Сопрограмма

Л

Литерал строковый 208
Лямбда-функция 136, 139

М

Максимум 19
Манипулятор 23
Массив C 119, 122
Минимум 19
Многозадачность кооперативная 25
Многопоточность 23
Модель памяти 24
Модификация типов 57
Мьютекс 296, 298

О

Обертка вокруг ссылки 41
Обертка
копируемо-конструируемая 41
Обертка
копируемо-присваиваемая 41
Обертка ссылочная
функциональная 19
Объект строковый 208, 227
Объект функциональный 136, 137
Операция модифицирующая 221, 228
Операция немодифицирующая 228
Отношение типа 56

П

Пара 38
 Переменная атомарная 23
 Переменная коллективная 24, 296
 Переменная потоково-локальная 24
 Переменная состояния 23
 Переменная условная 24, 307
 Перемещение 139
 Перемещение значений 19
 Перестановка значений местами 19
 Политика исполнения 21, 200
 Последовательность смежно
 расположенных элементов 21
 Поток ввода 245
 Поток вывода 245
 Поток данных 21, 134
 Поток инструкций 23
 Пояс часовой 19
 Предикат 30, 136, 142, 309
 Признак типа 19, 52
 Приложение
 событийно-управляемое 25
 Проекция 200
 Производительность 100, 106, 108

Р

Размах
 динамический 119, 123
 статический 119, 123
 Распределение случайных чисел 202

С

Свойство типа 55
 Семантика копирования 33
 Семантика перемещения 33
 Семафор 24, 310
 Сигнал остановки 24, 293
 Символ конца файла 216

Синхронизация 321
 Система файловая 269
 Сопрограмма 24, 195, 323
 Спецификатор хранения 306
 Сравнение целых чисел
 безопасное 19
 Ссылка универсальная 34
 Ссылка циклическая 51
 Сторожок 190, 191

Т

Тип агрегатный 79
 Тип первичный 53
 Тип пользовательский 104, 109, 305
 Тип составной 54
 Точка срединная 19

У

Удалитель специальный 47
 Указатель с размером 119, 122
 Указатель умный 19, 43
 Упорядочивание строго слабое 104

Ф

Функтор 137
 Функция 136, 137
 Функция стандартная
 математическая 22
 Функция форматирующая 262

Х

Хеш-функция 109

Ш

Шаблон вариативный 58

Книги издательства «ДМК Пресс» можно купить оптом и в розницу
на складе издательства по адресу:

**г. Москва, ул. Электродная, д. 2, стр. 12, офис 7,
тел. +7 (499) 322-19-38,**

а также заказать на сайте **www.dmkpress.com**
с доставкой в любой регион РФ

Райнер Гримм

**Стандартная библиотека C++
в примерах и пояснениях**

Все, что должен знать каждый профессиональный
программист на C++ о стандартной библиотеке

Главный редактор	<i>Мовчан Д. А.</i>
Зам. главного редактора	<i>Яценков В. С.</i> editor@dmkpress.com
Перевод с английского	<i>Логунов А. В.</i>
Корректор	<i>Синяева Г. И.</i>
Верстка	<i>Паранская Н. В.</i>
Дизайн обложки	<i>Мовчан А. Г.</i>

Формат 70×100¹/₁₆.
Печать цифровая. Усл. печ. л. 27.3.
Тираж 100 экз.

Веб-сайт издательства: **www.dmkpress.com**

О стандартной библиотеке C++ кратко и ясно

Книга представляет собой компактный справочник по стандартной библиотеке языка программирования C++, обновленной до версии стандарта C++23. В ней изложена вся необходимая информация, которую должен знать о стандартной библиотеке профессиональный программист на C++.

С помощью этого справочника вы:

- познакомитесь с различными библиотеками в рамках общей стандартной библиотеки и стандартной библиотеки шаблонов C++;
- овладеете мощным арсеналом вспомогательных инструментов, обобщенных типов, шаблонных классов и шаблонных функций;
- разберетесь в особенностях контейнеров и их интерфейсов, строковых объектов и регулярных выражений;
- научитесь работать с потоками ввода-вывода данных и форматированием;
- разберетесь в деталях конкурентной обработки потоков инструкций с использованием механизмов синхронизации;
- познакомитесь с устройством фреймворка сопрограмм.



Райнер Гримм – архитектор программного обеспечения, тимлид и наставник. С 2016 года является независимым преподавателем и проводит семинары по языкам C++ и Python. Опубликовал несколько книг на разных языках о современном языке C++ и, в частности, о конкурентности.

MC++
www.ModernesCpp.de

ДМК
издательство
www.дмк.рф



ISBN 978-5-93700-380-5

