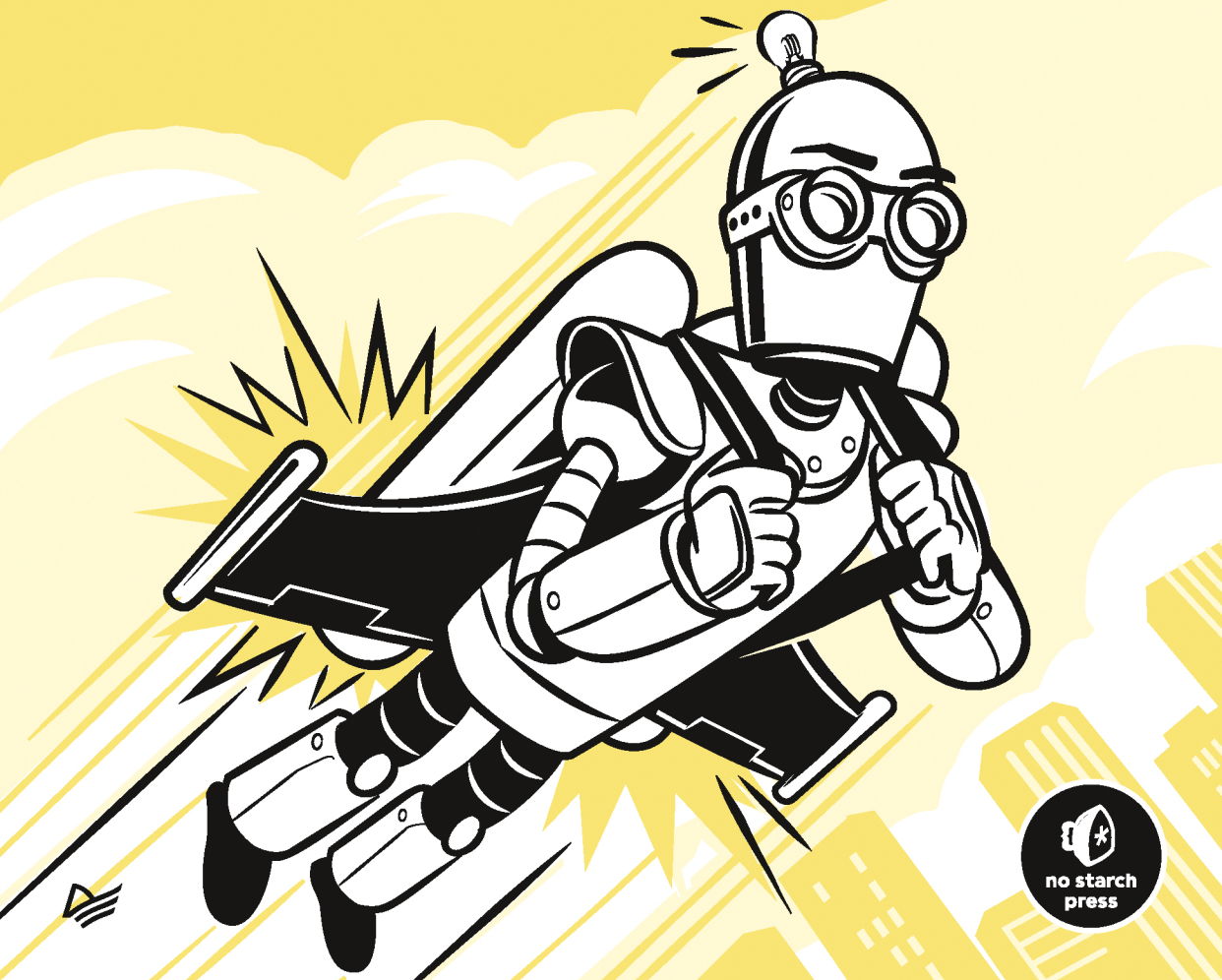
C++ ДЛЯ ПРОФИ

МОЛНИЕНОСНЫЙ СТАРТ

ДЖОШ ЛОСПИНОЗО

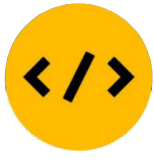


no starch press

C++ CRASH COURSE

A Fast-Paced Introduction

by Josh Lospinoso



no starch press
San Francisco

ДЖОШ ЛОСПИНОЗО

C++ ДЛЯ ПРОФИ

МОЛНИЕНОСНЫЙ СТАРТ

ПИТЕР®

Санкт-Петербург • Москва • Екатеринбург • Воронеж
Нижний Новгород • Ростов-на-Дону
Самара • Минск

2021

ББК 32.973.2-018.1
УДК 004.43
Л79

Лоспинозо Джош

Л79 С++ для профи. — СПб.: Питер, 2021. — 816 с.: ил. — (Серия «Для профессионалов»).

ISBN 978-5-4461-1730-7

С++ — популярный язык для создания ПО. В руках увлеченного программиста С++ становится прекрасным инструментом для создания лаконичного, эффективного и читаемого кода, которым можно гордиться. «С++ для профи» адресован программистам среднего и продвинутого уровней, вы продеретесь сквозь тернии к самому ядру С++. Часть 1 охватывает основы языка С++ — от типов и функций до жизненного цикла объектов и выражений. В части 2 представлена стандартная библиотека С++ и библиотеки Boost. Вы узнаете о специальных вспомогательных классах, структурах данных и алгоритмах, а также о том, как управлять файловыми системами и создавать высокопроизводительные программы, которые обмениваются данными по сети.

16+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ББК 32.973.2-018.1
УДК 004.43

ISBN 978-1593278885 англ.

ISBN 978-5-4461-1730-7

Краткое содержание

Оглавление

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

int main() {
    auto i{ 0x01B99644 };
    std::string x{ " DFaeeillnor" };
    while (i--) std::next_permutation(x.begin(), x.end());
    std::cout << x;
}
```

Об авторе

Джош Лоспинозо (Josh Lospinoso) — доктор философии и предприниматель, прослуживший 15 лет в армии США. Джош — офицер, занимающийся вопросами кибербезопасности. Написал десятки программ для средств информационной безопасности и преподавал C++ начинающим разработчикам. Выступает на различных конференциях, является автором более 20 рецензируемых статей и стипендиатом Родса, а также имеет патент. В 2012 году стал соучредителем успешной охранной компании. Джош ведет блог и активно участвует в разработке ПО с открытым исходным кодом.

О научном редакторе

Кайл Уиллмон (Kyle Willmon) — разработчик информационных систем с 12-летним опытом в C++. В течение 7 лет работал в сообществе по информационной безопасности, используя C++, Python и Go в различных проектах. В настоящее время является разработчиком в команде Sony Global Threat Emulation.

Предисловие

«C++ — сложный язык». Эту репутацию C++ заработал за несколько десятилетий, причем не совсем справедливо. Часто это утверждение считают поводом не изучать C++ или причиной, по которой стоит выбрать другой язык программирования. Такие аргументы трудно обосновать, ибо основная предпосылка неверна: C++ — не сложный язык. Первая проблема C++ — это его репутация. Вторая — отсутствие качественных учебных материалов для его изучения.

Сам язык за последние четыре десятилетия эволюционировал из языка C. Он начинался как ответвление C (с небольшими дополнениями) и предкомпилятор под названием Cfront, компилирующий ранний код C++ в код C, который затем обрабатывался с помощью компилятора C. Отсюда и название Cfront — «перед C». Спустя несколько лет оказалось, что это решение слишком ограничивает язык, и была предпринята работа по созданию фактического компилятора. Компилятор, написанный Бьёрном Страуструпом (Bjarne Stroustrup) (изобретатель языка), мог компилировать программу на C++ отдельно. Другие компании также были заинтересованы в продолжении базовой поддержки C и создали свои собственные компиляторы C++, в основном совместимые с Cfront или более новым компилятором.

Такой подход оказался несостоятельным, так как язык был непереносимым и абсолютно несовместимым между компиляторами, не говоря уже о том, что хранение всех решений и указаний в руках одного человека не способствовало созданию международного стандарта между компаниями — для этого существуют стандартные процедуры и организации, которые ими управляют. Таким образом, C++ стал стандартом ISO. После нескольких лет разработки в 1998 году вышел первый официальный стандарт C++, и люди возрадовались.

Но радовались недолго: хотя C++ 98 и был хорошим определением, он включал в себя несколько новых неожиданных разработок и имел пару функций, которые между собой весьма странно взаимодействовали. В некоторых случаях сами функции были хорошо написаны, но взаимодействие между общими функциями просто отсутствовало: например, возможность иметь имя файла в виде `std::string` и затем открывать файл с ним.

Другим запоздалым дополнением стала поддержка шаблонов — основная базовая технология, поддерживающая библиотеку стандартных шаблонов, одну из самых важных частей C++ на сегодняшний день. Только после ее выпуска люди обнаружили, что она сама по себе является полной по Тьюрингу и что многие сложные конструкции могут быть вычислены во время компиляции. Это значительно расширило возможности авторов библиотек при написании универсального кода,

который мог бы произвольно обрабатывать сложные выводы, что было не похоже на то, что могли делать другие существующие в то время языки.

Последняя сложность заключалась в том, что хотя C++ 98 и был хорош, многие компиляторы не подходили для реализации шаблонов. Два главных компилятора того времени, GNU GCC 2.7 и Microsoft Visual C++ 6.0, не смогли выполнить двухэтапный поиск имени, требуемый шаблонами. Единственным способом реализовать это право стало бы полное переписывание компилятора...

GNU пытался и дальше добавлять функции в существующую кодовую базу, но в конце концов перешел к переписыванию компилятора в версии 2.95. Долгие годы новых функций или релизов не было, и многие были недовольны этим. Некоторые компании взяли кодовую базу и попытались продолжить ее разработку, создав 2.95.2, 2.95.3 и 2.96 — все три версии известны своей нестабильностью. Наконец, вышел полностью переписанный GCC 3.0. Первоначально этот релиз был не очень успешным. Он собирал шаблоны и код C++ намного лучше, чем когда-либо делал 2.95, но не собирал ядро Linux в работающий двоичный файл. Сообщество Linux возражало против изменения своего кода для адаптации к новому компилятору, настаивая на том, что сам компилятор не работает. В конце концов, ко времени версии 3.2 сообщество сдалось и мир Linux снова сконцентрировался вокруг GCC 3.2 и выше.

Microsoft как мог старался избежать переписывания своего компилятора. Он добавлял граничные случаи к граничным случаям и эвристику, чтобы угадать, должно ли что-то быть разрешено в первом или втором проходе поиска имени шаблона. Такой подход почти сработал, но библиотеки, написанные в начале 2010-х годов, показали, что никакого возможного способа заставить их всех работать не было — даже с изменениями исходного кода. Корпорация Microsoft наконец-то переписала свой парсер и выпустила обновленную версию в 2018 году, но многие пользователи новый парсер не активировали. В 2019 году новый парсер был окончательно включен по умолчанию для новых проектов.

Ранее, в 2011 году, произошло важное событие: релиз C++ 11. После выхода C++ 98 были предложены и разработаны новые важные функции. Но из-за того, что одна функция работала не совсем так, как ожидалось, новый релиз был отложен до 2009 года. За это время были предприняты попытки заставить его работать с новой функцией. В 2009 году она была окончательно удалена, все остальное было исправлено, а версия C++ 1998 года была обновлена. Было добавлено множество новых функций и улучшений библиотеки. Компиляторы снова медленно догоняли релизы, и большинство компиляторов смогло скомпилировать большую часть C++ 11 только к концу 2013 года.

Комитет C++ извлек уроки из собственного раннего провала и представил план создания нового релиза каждые три года. План состоял в том, чтобы продумать и протестировать новые функции в течение одного года, как следует интегрировать их в следующем, стабилизировать и официально выпустить на третий год, а затем повторять этот процесс каждые три года. C++ 11 был первым экземпляром,

а 2014 год — предполагаемым годом для второго выпуска. К своей чести, комитет выполнил то, что обещал, подготовив серьезное обновление по сравнению с C++ 11 и сделав функционал C++ 11 намного более удобным, чем ранее. В большинстве случаев, где были введены осторожные ограничения, их перенесли на то, что тогда считалось приемлемым, в частности на ограничения вокруг `constexpr`.

Авторы компиляторов, которые все еще пытались заставить работать все функции C++ 11 хорошо, поняли, что нужно сменить темп, иначе они рискуют остаться позади. К 2015 году все компиляторы поддерживали практически всё в C++ 14, что можно счесть за подвиг, учитывая историю C++ 98 и C++ 11. Это также повлияло на возобновление участия в комитете по C++ всех основных авторов компиляторов — если вы знаете о функции до ее релиза, то можете стать ведущим компилятором, поддерживающим ее. Если вы обнаружите, что определенная функция не соответствует дизайну компилятора, то можете повлиять на комитет C++, настроив функцию так, чтобы ее было легче поддерживать, — таким образом вы дадите людям шанс использовать ее как можно раньше.

C++ переживает второе рождение, что началось примерно в 2011 году, когда был представлен C++ 11 и использован добавленный им стиль программирования «Modern C++». Однако он улучшился только потому, что все идеи из C++ 11 были точно настроены в C++ 14 и C++ 17, и компиляторы стали полностью поддерживать все ожидаемые функции. Более того, скоро будет выпущен новый стандарт для C++ 20 и все компиляторы в его самых современных версиях уже поддерживают его основные части.

Современный C ++ позволяет разработчикам избавиться от большинства первоначальных проблем, связанных с попытками сначала изучить C, затем C ++ 98, затем C ++ 11, а затем отучиться от всех исправленных частей C и C ++ 98. Большинство курсов начиналось с истории C++, так как было важно понять, почему что-то в нем кажется таким странным. Что касается этой книги, я включил эту информацию в предисловие, потому что Джош по праву не затронул ее.

Больше не нужно знать эту историю, чтобы начать изучать C++. Современный стиль C++ позволяет полностью пропустить ее и писать хорошо спроектированные программы, зная только основные принципы C++. Лучшее время для изучения C++ — сейчас.

Теперь вернемся к раннее упомянутому моменту — отсутствию качественных образовательных возможностей и материалов для изучения C++. В самом комитете C++ предоставляется высококачественное обучение C++ — есть учебная группа, занимающаяся исключительно преподаванием C++!

В отличие от всех других книг по C++, которые я читал, эта книга учит основам и принципам. Она учит принципам мышления, которые позволят решать задачи и учитывать возможности библиотеки стандартных шаблонов. Возможно, результаты вы получите не сразу, но когда увидите, как компилируются и запускаются первые программы, а вы полностью понимаете, как работает C++, то степень удовлетворенности будет выше. Эта книга включает в себя даже такие темы, которые

большинство книг по C++ игнорирует: настройка среды и тестирование кода перед запуском целой программы.

Наслаждайтесь чтением этой книги, делайте упражнения. А я желаю вам удачи в путешествии по C++!

Питер Биндельс (Peter Bindels),
главный инженер-программист, TomTom

Благодарности

Прежде всего благодарю свою семью за предоставленное мне творческое пространство. Написание книги заняло вдвое больше времени, чем я планировал, и я неизмеримо благодарен вам за ваше терпение.

Я в долгу перед Кайлом Уилмоном и Аароном Бреем (Aaron Bray), которые научили меня C++; Тайлеру Ортману (Tyler Ortman), который вел эту книгу от замысла до реального результата; Биллу Поллоку (Bill Pollock), который «причесал» мой стиль изложения; Крису Кливленду (Chris Cleveland), Патрику Де Хусто (Patrick De Justo), Энн Мари Уокер (Anne Marie Walker), Энни Чой (Annie Choi), Мег Снирингер (Meg Sneeringer) и Райли Хоффман (Riley Hoffman), чья первоклассная работа над текстом принесла огромную пользу этой книге, и многим первым читателям, оставившим неоценимые комментарии к сырым главам.

И наконец, я благодарю Джеффа Лоспинозо (Jeff Lospinoso), который завещал своему наивному десятилетнему племяннику зачитанную, испачканную кофе верблюжью книгу[1], с которой все и началось.

[1] *Уолл Л., Кристиансен Т., Орвант Д.* Программирование на Perl / Пер. с англ. — СПб.: Символ Плюс, 2002. — 1152 с.

От издательства

Ваши замечания, предложения, вопросы отправляйте по адресу comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На веб-сайте издательства www.piter.com вы найдете подробную информацию о наших книгах.

Введение

Хватай кисть и рисуй вместе с нами.

Боб Росс

Спрос на программирование систем огромен. Возможно, никогда еще не было лучшего времени, чтобы стать системным программистом, учитывая распространенность веб-браузеров, мобильных устройств и интернета вещей. Эффективный, поддерживаемый и правильный код желателен во всех случаях, и я твердо убежден, что C++ является подходящим языком для работы *в целом*.

В руках опытного программиста C++ может создавать меньший, более эффективный и более читаемый код, чем любой другой язык системного программирования на планете. Это язык, который стремится к абстракциям с нулевыми издержками. Вы сможете быстро программировать и вместе с простым прямым сопоставлением с аппаратным обеспечением иметь низкоуровневый контроль в любой момент. Программируя на C++, вы стоите на плечах гигантов, которые десятилетиями создавали невероятно мощный и гибкий язык.

Огромным преимуществом изучения C++ является бесплатный доступ к стандартной библиотеке C++, *stdlib*. Stdlib состоит из трех взаимосвязанных частей: контейнеров, итераторов и алгоритмов. Если вы когда-либо писали свой собственный алгоритм быстрой сортировки вручную или программировали системный код и столкнулись с переполнением буфера, висячими указателями, освобождением после использования и двойными освобождениями, то stdlib придется вам по душе. Она предоставляет непревзойденную комбинацию безопасности, правильности и эффективности типов. Кроме того, вам понравится то, насколько компактным и выразительным может быть код.

В основе модели программирования C++ лежит *жизненный цикл объекта*, который дает надежные гарантии того, что ресурсы, используемые программой: файлы, память и сетевые сокеты, освобождаются правильно даже в случае возникновения ошибок. При эффективном использовании исключения могут убрать из кода большое количество помех, связанных с проверкой условий. Кроме того, семантика перемещения/копирования обеспечивает безопасность, эффективность и гибкость для управления владением ресурсами так, как не могли предоставить ранние языки системного программирования, такие как C.

C++ — живой язык. Спустя более 30 лет комитет Международной организации по стандартизации (ISO) по C++продолжает регулярно вносить улучшения в язык. За последнее десятилетие было выпущено несколько обновлений стандарта: C++11, C++14 и C++17 — в 2011, 2014 и 2017 годах соответственно. В 2020 году вышел C++20.

Когда я использую термин *современный* C++, то имею в виду последний релиз C++, который включает в себя функции и парадигмы, представленные в этих дополнениях. Эти обновления внесли серьезные улучшения в язык, усовершенствовав его выразительность, эффективность, безопасность и общее удобство использования. По некоторым параметрам язык никогда не был более популярным, и он не исчезнет в ближайшее время. Если вы решите инвестировать в изучение C++, он будет приносить дивиденды долгие годы.

Об этой книге

Современные программисты на C++ имеют доступ к ряду очень качественных книг, например «Эффективный современный C++» Скотта Мейерса[1] и «Язык программирования C++» Бьёрна Страуструпа, 4-е издание[2]. Однако эти книги написаны для достаточно продвинутых программистов. Доступны также некоторые вводные тексты о C++, но они часто пропускают важные детали, потому что ориентированы на абсолютных новичков в программировании. Опытному программисту непонятно, где можно погрузиться в язык C++.

Я предпочитаю изучать сложные темы осознанно, выстраивая концепции из их основных элементов. Язык C++ имеет пугающую репутацию, потому что его фундаментальные элементы тесно связаны друг с другом, что затрудняет построение полной картины языка. Когда я изучал C++, то изо всех сил пытался сосредоточиться на языке, перескакивая от книг к видео и измученным коллегам. Поэтому и написал такую книгу, которую сам хотел бы иметь пять лет назад.

Кому будет интересна эта книга?

Эта книга предназначена для программистов среднего и продвинутого уровня, уже знакомых с основными концепциями программирования. Если у вас нет опыта в программировании *систем*, ничего страшного. Опытным программистам приложений издание также будет полезно.

---

[1] *Мейерс С.* М45 Эффективный и современный C++: 42 рекомендации по использованию C++ 11 и C++14 / Пер. с англ. — М.: ООО «ИЛ. Вильямс», 2016. — 304 с. — *Примеч. ред.*

[2] На русском языке можно найти перевод специального издания книги: *Страуструп Б.* Язык программирования C++. Специальное издание. М: Бином, 2020. — 1136 с. — *Примеч. ред.*

ПРИМЕЧАНИЕ

Если вы опытный программист на C или начинающий системный программист, задающийся вопросом, стоит ли вкладывать средства в изучение C++, обязательно прочитайте «Увертюру для C-программистов».

Структура книги

Книга разделена на две части. Часть I охватывает основной язык C++. Здесь не будет истории языка C++ (от старого стиля C++ 98 до современного C++ 11/14/17). Вы изучите непосредственно идиоматический современный C++. Часть II познакомит вас с миром стандартной библиотеки C++ (stdlib), где вы узнаете самые важные понятия.

Часть I. Основы языка C++

- **Глава 1. Создаем и запускаем.** Эта вводная глава поможет настроить среду разработки C++. Вы скомпилируете и запустите свою первую программу и узнаете, как ее отладить.
- **Глава 2. Типы.** Здесь вы познакомитесь с системой типов в C++. Вы узнаете об основных типах — фундаменте, на котором строятся остальные типы. Затем узнаете о старых простых типах данных и полнофункциональных классах, изучите роль конструкторов, инициализации и деструкторов.
- **Глава 3. Ссылочные типы.** В этой главе вы познакомитесь с объектами, которые хранят адреса других объектов в памяти. Эти типы являются краеугольным камнем многих важных паттернов программирования и позволяют создавать гибкий и эффективный код.
- **Глава 4. Жизненный цикл объекта.** Обсуждение инвариантов класса и конструктора продолжается в контексте продолжительности хранения. Деструктор вводится вместе с парадигмой инициализации ресурсов (RAII). Вы узнаете об исключениях и о том, как они реализуют инварианты классов и дополняют RAII. После обсуждения семантики перемещения и копирования вы узнаете, как их реализовать с помощью конструкторов и операторов присваивания.
- **Глава 5. Полиморфизм во время выполнения.** Рассказывает об интерфейсах — концепции программирования, которая позволяет писать код, являющийся полиморфным во время выполнения. Вы изучите базис наследования и композиции объектов, которые лежат в основе использования интерфейсов в C++.
- **Глава 6. Полиморфизм во время компиляции.** В этой главе представлены шаблоны — языковая конструкция, позволяющая писать полиморфный код. Вы также изучите концепции, которые будут добавлены в будущем выпуске C++, и именованные функции преобразования, позволяющие преобразовывать объекты из одного типа в другой.

- **Глава 7. Выражения.** Здесь речь пойдет об операндах и операторах. Хорошо разбираясь в типах, жизненном цикле объектов и шаблонах, вы будете готовы погрузиться в основные компоненты языка C++. Инструкции — первая остановка на этом пути.
- **Глава 8. Инструкции.** В этой главе рассматриваются элементы, составляющие функции. Вы узнаете об операторах выражений, составных операторах, операторах объявлений, операторах итерации и операторах перехода.
- **Глава 9. Функции.** В последней главе части I подробно рассматривается группировка утверждений в единицы работы. Вы узнаете подробности об определениях функций, типах возвращаемых данных, разрешении перегрузки, переменных функциях, переменных шаблонах и указателях функций. Вы также узнаете, как создавать вызываемые пользовательские типы, используя оператор вызова функции и лямбда-выражения. Вы изучите `std::function`, класс, который предоставляет единый контейнер для хранения вызываемых объектов.

Часть II. Библиотеки и фреймворки C++

- **Глава 10. Тестирование.** Эта глава познакомит с удивительным миром фреймворков юнит-тестирования и имитации (мокирования). Вы попрактикуетесь в разработке через тестирование на примере автономной системы вождения, изучая такие фреймворки, как Boost Test, Google Test, Google Mock и другие.
- **Глава 11. Умные указатели.** Здесь описываются специальные служебные классы, которые предоставляет stdlib для управления владением динамическими объектами.
- **Глава 12. Утилиты.** Здесь вы найдете обзор типов, классов и функций, имеющихся в библиотеках stdlib и Boost для решения распространенных проблем программирования. Вы узнаете о структурах данных, числовых функциях и генераторах случайных чисел.
- **Глава 13. Контейнеры.** В этой главе рассматриваются специальные структуры данных из библиотек Boost и stdlib, которые помогают организовать данные. Вы узнаете о контейнерах последовательностей, ассоциативных контейнерах и неупорядоченных ассоциативных контейнерах.
- **Глава 14. Итераторы.** Это интерфейс между контейнерами, о которых вы узнали в предыдущей главе, и строками из следующей главы. Вы узнаете о различных видах итераторов и о том, как их дизайн обеспечивает невероятную гибкость.
- **Глава 15. Строки.** В этой главе рассказывается, как обрабатывать данные на естественном языке в одном семействе контейнеров. Также узнаете о специальных средствах, встроенных в строки, которые позволяют решать общие задачи.
- **Глава 16. Потоки.** Здесь вы познакомитесь с основной концепцией, лежащей в основе операций ввода и вывода. Вы узнаете, как обрабатывать потоки ввода и вывода с помощью форматированных и неформатированных операций,

а также как использовать манипуляторы. Вы узнаете, как читать и записывать данные из файлов и в файлы.

- **Глава 17. Файловые системы.** Здесь вы получите обзор возможностей stdlib для управления файловыми системами. Вы узнаете, как создавать и управлять путями, проверять файлы и каталоги и перечислять структуры каталогов.
- **Глава 18. Алгоритмы.** Эта глава представляет собой краткий справочник по десяткам задач, которые можно легко решить с помощью stdlib. Вы узнаете о впечатляющих возможностях высококачественных алгоритмов.
- **Глава 19. Конкурентность и параллелизм.** В этой главе рассказывается о некоторых простых методах конкурентного программирования, которые являются частью stdlib. Вы узнаете о фьючерсах, мьютексах, условных переменных и атомарности.
- **Глава 20. Сетевое программирование с помощью Boost Asio.** Здесь вы узнаете, как создавать высокопроизводительные программы, которые взаимодействуют по сетям. Вы увидите, как использовать Boost Asio с блокирующим и неблокирующим вводом и выводом.
- **Глава 21. Создание приложений.** Последняя глава завершает книгу обсуждением нескольких важных тем. Вы узнаете о средствах поддержки программ, которые позволяют подключиться к жизненному циклу приложения. Также вы узнаете о Boost ProgramOptions — библиотеке, которая делает написание консольных приложений, принимающих ввод данных пользователем, простым.

ПРИМЕЧАНИЕ

На сайте ccc.codes вы можете найти фрагменты кода из этой книги.

Увертюра для С-программистов

Артур Дент: «Что с ним не так?»

Хиг Хуртенфлюрст: «Его ноги не подходят под размер его ботинок».

Дуглас Адамс, «Автостопом по Галактике. Вопль одиннадцатый»

Это предисловие предназначено для опытных программистов на С, которые сомневаются, стоит ли читать эту книгу. Программисты, не владеющие С, могут эту часть пропустить.

Бьёрн Страуструп разработал С++ из языка программирования С. Хотя С++ не полностью совместим с языком С, хорошо написанные С-программы часто также являются допустимыми С++-программами. Например, каждый пример из «Языка программирования Си»[1] (The C Programming Language) Брайана Кернигана (Brian Kernighan) и Денниса Ритчи (Dennis Ritchie) — допустимая программа на С++.

Одна из главных причин распространенности С в сообществе системного программирования состоит в том, что С позволяет писать на более высоком уровне абстракции, чем программирование на ассемблере. Это приводит к созданию более чистого, менее подверженного ошибкам и проще обслуживаемого кода.

Как правило, системные программисты не хотят платить за удобство программирования, поэтому С придерживается принципа нулевых издержек: *вы не платите за то, что не используете*. Строго типизированная система является ярким примером абстракции с нулевыми издержками. Она используется только во время компиляции для проверки правильности программы. Со временем компиляция типов исчезнет, а в коде готовой сборки не останется никаких следов системы типов.

Будучи потомком С, язык С++ также очень серьезно относится к абстракции без издержек и прямому сопоставлению с аппаратным обеспечением. Это обязательство выходит за рамки возможностей языка С, которые поддерживает С++. Все, что С++ строит поверх С, включая новые языковые возможности, поддерживает эти принципы. Отходы от них сделаны очень осознанно. На самом деле некоторые

[1] *Керниган Б., Ритчи Д.* Язык программирования Си / Пер. с англ. 3-е изд., испр. — СПб.: Невский Диалект, 2001.— 352 с. — *Примеч. ред.*

функции C++ несут еще меньше издержек, чем соответствующий код C. Ключевое слово `constexpr` является одним из таких примеров. Оно указывает компилятору вычислить выражение во время компиляции (если это возможно), как показано в листинге 1.

Листинг 1. Программа, демонстрирующая constexpr

```
#include <cstdio>

constexpr int isqrt(int n) {
  int i=1;
  while (i*i<n) ++i;
  return i-(i*i!=n);
}

int main() {
  constexpr int x = isqrt(1764); ❶
  printf("%d", x);
}
```

Функция `isqrt` вычисляет квадратный корень аргумента `n`. Начиная с `1`, функция увеличивает локальную переменную `i` до тех пор, пока `i*i` не станет больше или равно `n`. Если `i*i == n`, возвращается `i`; в противном случае возвращается `i-1`. Обратите внимание, что вызов `isqrt` имеет буквальное значение, поэтому компилятор теоретически может вычислить результат, который будет принимать только одно значение ❶.

Компиляция листинга 1 в GCC 8.3 для x86-64 с `-O2` приведет к сборке из листинга 2.

Листинг 2. Сборка, созданная после компиляции листинга 1

```
.LC0:
        .string "%d"
main:
        sub rsp, 8
        mov esi, 42 ❶
        mov edi, OFFSET FLAT:.LC0
        xor eax, eax
        call printf
        xor eax, eax
        add rsp, 8
        ret
```

Важным результатом здесь является вторая инструкция в `main` ❶; вместо того чтобы вычислять квадратный корень из `1764` во время выполнения, компилятор оценивает его и выводит инструкции для обработки `x` как `42`. Конечно, можно вычислить квадратный корень с помощью калькулятора и вставить результат вручную, но использование `constexpr` имеет много преимуществ. Этот подход может снизить вероятность многих ошибок, связанных с ручным копированием и вставкой, и сделать код более выразительным.

ПРИМЕЧАНИЕ

Если вы не знакомы с ассемблером x86, почитайте второе издание «Искусства ассемблера» Рэндалла Хайда (The Art of Assembly Language, Randall Hyde) и «Профессиональный язык ассемблера» Ричарда Блума (Professional Assembly Language, Richard Blum).

Обновление до Super C

Современные компиляторы C++ учтут большинство ваших навыков программирования на C. Это облегчает использование некоторых тактических изысков, которые предоставляет язык C++, но в то же время обходит более глубокие темы языка. Такой стиль C++ — назовем его *Super* C — важно обсудить по нескольким причинам. Во-первых, опытные программисты на C могут сразу получить выгоду от применения простых тактических понятий C++ к своим программам. Во-вторых, Super C *не* является идиоматическим C++. Простое добавление ссылок и экземпляров `auto` в C-программы может сделать код более надежным и читаемым, но для того, чтобы в полной мере воспользоваться этими преимуществами, стоит изучить другие концепции. В-третьих, в некоторых строгих средах (например, встроенное программное обеспечение, некоторые ядра операционных систем и гетерогенные вычисления) доступные цепочки инструментов имеют неполную поддержку C++. В таких ситуациях можно извлечь выгоду по крайней мере из некоторых идиом C++ и Super C, скорее всего, будет поддерживаться. В этом разделе рассматриваются некоторые концепции Super C, которые можно тут же применить к своему коду.

ПРИМЕЧАНИЕ

Некоторые C-поддерживаемые конструкции не будут работать в C++. Смотрите раздел ссылок на сайте этой книги: ccc.codes.

Перегрузка функций

Рассмотрим следующие функции преобразования из стандартной библиотеки C:

```
char* itoa(int value, char* str, int base);
char* ltoa(long value, char* buffer, int base);
char* ultoa(unsigned long value, char* buffer, int base);
```

Эти функции служат одной и той же цели: преобразуют целочисленный тип в строку в стиле C. В языке C каждая функция должна иметь уникальное имя. Но в C++ функции могут совместно использовать имена при наличии различий в их аргументах; это называется *перегрузкой функции* (*function overloading*). Перегрузку функций можно использовать для создания своих собственных функций преобразования, как показано в листинге 3.

Листинг 3. Вызов перегруженных функций

```
char* toa(int value, char* buffer, int base) {
  --пропуск--
}

char* toa(long value, char* buffer, int base)
  --пропуск--
}

char* toa(unsigned long value, char* buffer, int base) {
  --пропуск--
}

int main() {
  char buff[10];
  int a = 1; ❶
  long b = 2; ❷
  unsigned long c = 3; ❸
  toa(a, buff, 10);
  toa(b, buff, 10);
  toa(c, buff, 10);
}
```

Тип данных первого аргумента в каждой из функций отличается, поэтому компилятор C++ имеет достаточно информации из аргументов, переданных в `toa`, для вызова правильной функции. Каждый вызов `toa` является уникальной функцией. Здесь были созданы переменные `a` ❶, `b` ❷ и `c` ❸, которые представляют собой различные типы объектов `int`, которые соответствуют одной из трех функций `toa`. Это удобнее, чем определять отдельно названные функции, потому что достаточно запомнить одно имя, а компилятор сам определит, какую функцию вызывать.

Ссылки

Указатели являются важной особенностью C (и, как следствие, системного программирования в целом). Они позволяют эффективно обрабатывать большие объемы данных, передавая адреса в памяти вместо фактических данных. Указатели одинаково важны и для C++, но в него были добавлены дополнительные функции безопасности, которые защищают от нулевых разыменований и непреднамеренных переназначений указателей. *Ссылки* значительно улучшают работу с указателями. Они похожи на указатели, но имеют некоторые ключевые отличия. Синтаксически ссылки отличаются от указателей по двум важным причинам. Во-первых, они должны быть объявлены знаком `&` вместо `*`, как показано в листинге 4.

Листинг 4. Код, показывающий, как объявлять функции с указателями и ссылками

```
struct HolmesIV {
  bool is_sentient;
  int sense_of_humor_rating;
};
void make_sentient(HolmesIV*); // Принимает указатель на HolmesIV HolmesIV
void make_sentient(HolmesIV&); // Принимает ссылку на HolmesIV
```

Во-вторых, взаимодействовать с членами нужно, используя оператор точки `.`, а не оператор стрелки `->`, как показано в листинге 5.

Листинг 5. Программа с операторами точки и стрелки

```
void make_sentient(HolmesIV* mike) {
  mike->is_sentient = true;
}

void make_sentient(HolmesIV& mike) {
  mike.is_sentient = true;
}
```

Под капотом ссылки эквивалентны указателям, потому что также являются абстракцией с нулевыми издержками. Компилятор в итоге выдает похожий код. Чтобы продемонстрировать это, рассмотрим результаты компиляции функций `make_sentient` в GCC 8.3 для x86-64 с `-O2`. В листинге 6 содержится сборка, созданная путем компиляции листинга 5.

Листинг 6. Сборка, сгенерированная путем компиляции листинга 5

```
make_sentient(HolmesIV*):
        mov     BYTE PTR [rdi], 1
        ret
make_sentient(HolmesIV&):
        mov     BYTE PTR [rdi], 1
        ret
```

Однако во время компиляции ссылки обеспечивают некоторую безопасность по сравнению с обычными указателями, потому что вообще-то они не могут быть нулевыми.

С помощью указателей можно добавить проверку `nullptr` в целях безопасности. Например, проверку можно провести в `make_sentient`, как показано в листинге 7.

Листинг 7. Рефакторинг функции make_sentient из листинга 5: теперь она выполняет проверку nullptr

```
void make_sentient(HolmesIV* mike) {
  if(mike == nullptr) return;
  mike->is_sentient = true;
}
```

Такая проверка не требуется при получении ссылки; однако это не означает, что ссылки всегда корректны. Рассмотрим следующую функцию:

```
HolmesIV& not_dinkum() {
  HolmesIV mike;
  return mike;
}
```

Функция `not_dinkum` возвращает ссылку, которая точно не равна `null`. Но она указывает на ненужную информацию в памяти (вероятно, в возвращаемом кадре

стека `not_dinkum`). Никогда так не делайте. Это приведет к беде, также известной как *неопределенное поведение во время выполнения* (*undefined runtime behavior*): может произойти сбой, ошибка или что-то совершенно неожиданное. Еще одной особенностью безопасности ссылок является то, что они не могут быть переопределены. Другими словами, после инициализации ссылки ее нельзя изменить так, чтобы она указывала на другой адрес в памяти, как показано в листинге 8.

Листинг 8. Пример, показывающий, что ссылки не могут быть повторно установлены

```
int main() {
  int a = 42;
  int& a_ref = a; ❶
  intb = 100;
  a_ref = b; ❷
}
```

`a_ref` была объявлена как ссылка на `int a` ❶. Не существует способа переопределить `a_ref` так, чтобы она указывала на другой `int`. Можно попробовать переопределить `a` с помощью `operator=` ❷, но это фактически устанавливает значение `a` равным значению `b` вместо установки `a_ref` в качестве ссылки на `b`. После запуска фрагмента и `a`, и `b` равны `100`, а `a_ref` по-прежнему указывает на `a`. В листинге 9 приведен эквивалентный код с использованием указателей.

Листинг 9. Программа, эквивалентная листингу 8, с указателями

```
int main() {
  int a = 42;
  int* a_ptr = &a; ❶
  intb = 100;
  *a_ptr = b; ❷
}
```

Здесь указатель объявляется с помощью * вместо & ❶. Вы присваиваете значение `b` памяти, на которую указывает `a_ptr` ❷. При использовании ссылок не потребуется никаких декораций слева от знака равенства. Но если опустить * в `*a_ptr`, компилятор пожалуется на попытку присвоить `int` типу указателя.

Ссылки — это просто указатели с дополнительными мерами предосторожности и небольшим количеством синтаксического сахара. Размещая ссылку слева от знака равенства, вы устанавливаете целевое значение указателя, равное правой стороне знака равенства.

Инициализация с помощью auto

В C часто требуется повтор информации о типе более одного раза. В C++ можно выразить информацию о типе переменной только один раз, используя ключевое слово `auto`. Компилятор будет знать тип переменной, потому что он знает тип зна-

чения, используемого для инициализации переменной. Рассмотрим следующие инициализации переменных в C++:

```
int x = 42;
auto y = 42;
```

Здесь и `x`, и `y` имеют тип `int`. Вы можете удивиться, узнав, что компилятор может определить тип `y`, но учтите, что 42 — это целочисленный литерал. При использовании `auto` компилятор выводит тип справа от знака равенства `=` и устанавливает такой же тип переменной. Поскольку целочисленный литерал имеет тип `int`, в этом примере компилятор определяет, что тип `y` также является типом `int`. Для такого простого примера это не кажется большим преимуществом, но давайте рассмотрим возможность инициализации переменной возвращаемым значением функции, как показано в листинге 10.

Листинг 10. Программа, инициализирующая переменную возвращаемым значением функции

```
#include <cstdlib>

struct HolmesIV {
  --пропуск--
};

HolmesIV* make_mike(int sense_of_humor) {
  --пропуск--
}

int main() {
  auto mike = make_mike(1000);
  free(mike);
}
```

Ключевое слово `auto` проще прочитать, и оно легче поддается рефакторингу, чем явное объявление типа переменной. Если вы свободно используете `auto` при объявлении функции, понадобится меньше работы для изменения типа возвращаемого значения `make_mike`. Использование `auto` усиливается более сложными типами, такими как типы, связанные с кодом, загружаемым из шаблона `stdlib`. Ключевое слово `auto` заставляет компилятор выполнять за вас всю работу по выводу типов.

ПРИМЕЧАНИЕ

В `auto` также можно добавить квалификаторы `const`, `volatile`, `&` и `*`.

Пространства имен и неявный typedef для struct, union и enum

C++ обрабатывает теги типа как неявные имена `typedef`. В C для использования `struct`, `union` или `enum` нужно назначить имя созданному типу с помощью ключевого слова `typedef`. Например:

```
typedef struct Jabberwocks {
  void* tulgey_wood;
  int is_galumphing;
} Jabberwock;
```

В кругах C++ над таким кодом можно только посмеяться. Поскольку ключевое слово `typedef` может быть неявным, C++ позволяет вместо него объявлять тип `Jabberwock` следующим образом:

```
struct Jabberwock {
  void* tulgey_wood;
  int is_galumphing;
};
```

Это удобнее и экономит время при наборе текста. Что произойдет, если также нужно определить функцию `Jabberwock`? Вообще, этого делать не стоит, потому что повторное использование одного и того же имени для типа данных и функции может породить путаницу. Но если без этого не обойтись, C++ позволяет объявлять пространство имен для создания различных областей действия для идентификаторов. Это помогает поддерживать чистоту пользовательских типов и функций, как показано в листинге 11.

Листинг 11. Использование пространств имен для устранения неоднозначности функций и типов с одинаковыми именами

```
#include <cstdio>

namespace Creature { ❶
  struct Jabberwock {
    void* tulgey_wood;
    int is_galumphing;
  };
}
namespace Func { ❷
  void Jabberwock() {
    printf("Burble!");
  }
}
```

В этом примере `struct Jabberwock` и функция `Jabberwock` теперь живут вместе в гармонии. Поместив каждый элемент в свое собственное пространство имен — `struct` в пространство имен `Creature` ❶ и функцию в пространство имен `Func`❷, — можно определить, какой из `Jabberwock` имеется в виду. Достичь такого устранения неоднозначности можно несколькими способами. Самое простое — определить имя по его пространству имен, например:

```
Creature::Jabberwock x;
Func::Jabberwock();
```

Также можно использовать директиву `using` для импорта всех имен в пространстве имен, поэтому больше не нужно будет использовать полное имя элемента. В листинге 12 используется пространство имен `Creature`.

Листинг 12. Использование пространства имен для ссылки на тип в пространстве имен Creature

```
#include <cstdio>

namespace Creature {
  struct Jabberwock {
    void* tulgey_wood;
    int is_galumphing;
  };
}

namespace Func {
  void Jabberwock() {
    printf("Burble!");
  }
}

using namespace Creature;  ❶

int main() {
  Jabberwock x; v ❷
  Func::Jabberwock();
}
```

`using namespace` ❶ позволяет опустить квалификацию `namespace` ❷. Но вам все равно нужен спецификатор в `Func::Jabberwock`, потому что он не является частью `Creature namespace`.

Использование `namespace` — пример идиоматического C++ и абстракции с нулевыми издержками. Как и остальные идентификаторы типа, `namespace` стирается компилятором при выдаче кода сборки. В больших проектах это невероятно полезно для разделения кода в разных библиотеках.

Смешение объектных файлов C и C++

Коды C и C++ могут мирно сосуществовать, если проявить предусмотрительность. Иногда компилятору C необходимо связать объектные файлы, выдаваемые компилятором C++ (и наоборот). Это реально, хотя и потребует немного усилий.

Со связыванием файлов возникают две проблемы. Во-первых, соглашения о вызове в кодах C и C++ могут потенциально не совпадать. Например, протоколы установки стека и регистров при вызове функции могут отличаться. Эти соглашения о вызове являются несоответствиями на уровне языка и обычно не связаны с тем, как вы написали функции. Во-вторых, компиляторы C++выдают символы, отличные от символов компиляторов C. Иногда редактор связей должен идентифицировать объект по имени. Компиляторы C++ помогают декорировать объект, связывая строку, называемую *декорированным именем* (*decorated name*), с объектом. Из-за перегрузок функций, соглашений о вызове и использования `namespace` компилятор должен кодировать дополнительную информацию о функции, помимо ее имени, посредством декорирования. Это сделано для того, чтобы редактор связей мог одно-

значно идентифицировать функцию. К сожалению, не существует стандарта того, как это оформление происходит в C++ (именно поэтому необходимо использовать одну и ту же цепочку инструментов и настройки при связывании между единицами трансляции (translation unit)). Редакторы связей C ничего не знают о декорировании имен C++, что может вызвать проблемы, если декорирование не подавляется при ссылках на код C в C++ (и наоборот).

Исправить это легко. Оберните код, который нужно скомпилировать, в стиле C, используя оператор `extern "C"`, как показано в листинге 13.

Листинг 13. Использование связей в стиле C

```
// header.h
#ifdef __cplusplus
extern "C" {
#endif
void extract_arkenstone();

struct MistyMountains {
  int goblin_count;
};
#ifdef __cplusplus
}
#endif
```

Этот заголовок может быть разделен на код C и C++ . Такой подход работает, потому что `__cplusplus` — это специальный идентификатор, который определяет компилятор C++ (но компилятор C этого не делает). Соответственно компилятор C видит код в листинге 14 после завершения препроцессинга (preprocessing). В листинге 14 показан оставшийся код.

Листинг 14. Код, оставшийся после обработки препроцессором листинга 13 в среде C

```
void extract_arkenstone();

struct MistyMountains {
  int goblin_count;
};
```

Это простой C-заголовок. Код между операторами `#ifdef __cplusplus` удаляется во время препроцессинга, поэтому обертка `extern "C"` остается незамеченной. Для компилятора C++ `__cplusplus` определен в `header.h`, поэтому он видит содержимое листинга 15.

Листинг 15. Код, оставшийся после обработки препроцессором листинга 13 в среде C++

```
extern "C" {
  void extract_arkenstone();

  struct MistyMountains {
    int goblin_count;
  };
}
```

И `extract_arkenstone`, и `MistyMountains` теперь заключены в `extern "C"`, поэтому компилятор знает, как использовать связь в стиле С. Теперь источник С может вызывать скомпилированный код С++, а источник С++ может вызывать скомпилированный код С.

Темы С++

В этом разделе вы познакомитесь с некоторыми основными темами, которые делают С++ лучшим языком системного программирования. Не беспокойтесь о деталях. Задача следующих подразделов — разжечь интерес.

Краткое изложение идей и повторное использование кода

Хорошо разработанный код С++элегантен и компактен. Рассмотрим переход от ANSI-C к современному С++ в следующей простой операции: перебор элементов в некотором массиве `v` с `n` элементами, как показано в листинге 16.

Листинг 16. Программа, показывающая несколько способов перебора массива

```
#include <cstddef>

int main() {
  const size_t n{ 100 };
  int v[n];

  // ANSI-C
  size_t i;
  for (i=0; i<n; i++) v[i]=0; ❶
  // C99
  for (size_t i=0; i<n; i++) v[i]=0; ❷

  // C++17
  for (auto& x : v) x = 0; ❸
}
```

В этом фрагменте кода показаны различные способы объявления циклов в ANSI-C, C99 и С++. Индексная переменная `i` в примерах ANSI-C ❶ и C99 ❷ является вспомогательной для того, что вы пытаетесь достичь, а именно для доступа к каждому элементу `v`. В версии С++ ❸ используется цикл `for`, *основанный на диапазоне*, который перебирает диапазон значений в `v`, скрывая детали того, как достигается итерация. Как и многие абстракции с нулевыми издержками в С++, эта конструкция позволяет сосредоточиться на значении, а не на синтаксисе. Циклы `for` на основе диапазона работают со многими типами, и можно даже заставить их работать с пользовательскими типами.

Что касается пользовательских типов, то они позволяют выражать идеи непосредственно в коде. Предположим, нужно сконструировать функцию `navigate_to`, кото-

рая сообщает гипотетическому роботу, что нужно перейти к некоторому положению с заданными координатами `x` и `y`. Рассмотрим следующую функцию-прототип:

```
void navigate_to(double x, double y);
```

Что такое `x` и `y`? Что за системы они представляют? Пользователь должен прочитать документацию (или, возможно, исходный код), чтобы это узнать. Сравните код выше с его улучшенной версией:

```
struct Position{
--пропуск--
};
void navigate_to(const Position& p);
```

Функция стала намного понятнее. Нет никакой двусмысленности в том, что принимает `navigate_to`. Пока существует правильно сконструированный экземпляр `Position`, вы точно знаете, как вызывать `navigate_to`. Беспокоиться о единицах, конверсиях и т. д. теперь стоит тому, кто создает класс `Position`.

К подобной ясности также можно приблизиться в C99/C11, используя указатель `const`, но C++ также делает возвращаемые типы компактными и выразительными. Предположим, нужно написать сопутствующую функцию `get_position` для робота, которая — как вы уже догадались — получает позицию. В C для этого существуют два варианта, как показано в листинге 17.

Листинг 17. API в стиле C для возврата пользовательского типа

```
Position* get_position(); ❶
void get_position(Position* p); ❷
```

В первом варианте вызывающая сторона отвечает за очистку возвращаемого значения ❶, которое, вероятно, подверглось динамическому распределению (хотя это неочевидно из кода). Вызывающая сторона ответственна за то, чтобы где-то выделить память для `Position` и передать полученный экземпляр в `get_position` ❷. Последний подход выражен в стиле C, но тут язык начинает мешать: вы просто пытаетесь получить объект позиции, при этом нужно беспокоиться о том, отвечает ли вызывающая или вызванная функция за выделение и освобождение памяти. C++ позволяет делать все это лаконично, возвращая пользовательские типы непосредственно из функций, как показано в листинге 18.

Листинг 18. Возвращение пользовательского типа по значению в C++

```
Position❶ get_position() {
  --пропуск--
}
void navigate() {
  auto p = get_position(); ❷
  // с этого момента можно использовать p
  --пропуск--
}
```

Поскольку `get_position` возвращает значение ❶, компилятор может аннулировать копию, поэтому создается впечатление, что автоматическая переменная `Position` была создана непосредственно ❷; не было издержек времени выполнения. Функционально вы находитесь на территории, очень похожей на проход в стиле С, как показано в листинге 17.

Стандартная библиотека C++

Стандартная библиотека C++ (stdlib) является основной причиной перехода из С. Она содержит высокопроизводительный общий код, который гарантированно доступен по умолчанию и при этом соответствует стандартам. Три важнейших компонента stdlib — это контейнеры, итераторы и алгоритмы.

Контейнеры — это структуры данных. Они отвечают за хранение последовательностей объектов. Они правильные, безопасные и (обычно) как минимум настолько же эффективные, как и то, что вы могли бы выполнить вручную. Это означает, что написать собственные версии этих контейнеров было бы сложнее и они не получились бы лучше, чем существующие контейнеры stdlib. Контейнеры четко разделены на две категории: *контейнеры последовательности* (*sequence containers*) и *ассоциативные контейнеры* (*associative containers*). Контейнеры последовательности концептуально похожи на массивы; они обеспечивают доступ к последовательностям элементов. Ассоциативные контейнеры содержат пары ключ/значение, поэтому элементы в контейнерах можно искать по ключу.

Алгоритмы stdlib являются функциями общего назначения для основных задач программирования, таких как подсчет, поиск, сортировка и преобразование. Подобно контейнерам, алгоритмы stdlib чрезвычайно качественны и широко применяются. Пользователям очень редко приходится реализовывать свою собственную версию, а использование алгоритмов stdlib значительно повышает производительность труда программиста, безопасность кода и удобочитаемость.

Итераторы связывают контейнеры с алгоритмами. Во многих случаях использования алгоритма stdlib данные, с которыми нужно работать, находятся в контейнере. Контейнеры предоставляют итераторы для обеспечения общего интерфейса, а алгоритмы используют итераторы, не позволяя программистам (включая разработчиков stdlib) реализовывать собственный алгоритм для каждого типа контейнера.

В листинге 19 показано, как отсортировать контейнер значений, используя несколько строк кода.

В фоновом режиме выполняется большое количество вычислений, но итоговый код компактен и выразителен. Сначала инициализируется контейнер `std::vector` ❶. *Векторы* — это динамические массивы stdlib. *Скобки инициализатора* (*initializer braces*) ({0, 1,...}) устанавливают начальные значения, содержащиеся в `x`. Доступ к элементам вектора можно получить так же, как к элементам массива, используя скобки (`[]`) и порядковый номер. Эта техника используется, чтобы установить первый элемент равным `21` ❷. Поскольку векторные массивы имеют динамический

размер, можно добавлять значения к ним, используя метод `push_back` ❸. Кажущийся магическим вызов `std::sort` демонстрирует мощь алгоритмов в `stdlib` ❹. Методы `x.begin()` и `x.end()` возвращают итераторы, которые `std::sort` использует для сортировки `x` на месте. Алгоритм сортировки отделен от вектора с помощью итераторов.

Листинг 19. Сортировка контейнера значений с использованием stdlib

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

int main() {
  std::vector<int> x{ 0, 1, 8, 13, 5, 2, 3 }; ❶
  x[0] = 21;  ❷
  x.push_back(1); ❸
  std::sort(x.begin(), x.end()); ❹
  std::cout << "Printing " << x.size() << " Fibonacci numbers.\n"; ❺
  for (auto number : x) {
    std::cout << number << std::endl; ❻
  }
}
```

Благодаря итераторам можно использовать другие контейнеры в stdlib аналогичным образом. Например, вместо вектора может быть использован список (двусвязный список в stdlib). Поскольку `list` также предоставляет итераторы с помощью методов `.begin()` и `.end()`, `sort` вызывается таким же образом для итераторов списка.

Кроме того, в листинге 19 используются потоки ввода/вывода. *Потоки ввода/вывода* (*iostreams*) — это механизм stdlib для выполнения буферизованного ввода и вывода. Оператор сдвига влево (`<<`) используется для потоковой передачи значения `x.size()` (количество элементов в `x`), некоторых строковых литералов и элемента Фибоначчи `number` в `std::cout`, который инкапсулирует стандартный вывод ❺ ❻. Объект `std::endl` является манипулятором ввода-вывода, который записывает `\n` и очищает буфер, гарантируя, что весь поток записывается в стандартный вывод перед выполнением следующей инструкции.

Теперь просто представьте полосу с препятствиями, которую нужно пройти, чтобы написать эквивалентную программу на C, и вы поймете, почему stdlib — такой ценный инструмент.

Лямбда-выражения

Лямбда-выражения (их также называют *анонимными функциями*) являются еще одной мощной языковой функцией, улучшающей локальность кода. В некоторых случаях нужно передавать указатели на функции, чтобы использовать указатель в качестве цели для вновь созданного потока или выполнять какое-либо преобразование для каждого элемента последовательности. Обычно для этого неудобно определять новую одноразовую функцию. Вот здесь и появляются лямбда-выражения. Лямб-

да-выражение — это новая настраиваемая функция, *определяемая в соответствии с другими параметрами вызова*. Рассмотрим следующую однострочную функцию, которая вычисляет количество четных чисел в `x`:

```
auto n_evens = std::count_if(x.begin(), x.end(),
                             [] (auto number) { return number % 2 == 0; });
```

Этот фрагмент использует алгоритм `count_if` из stdlib для подсчета четных чисел в `x`. Первые два аргумента для `std::count_if` соответствуют `std::sort`; это итераторы, которые определяют диапазон, в котором будет работать алгоритм. Третий аргумент — лямбда-выражение. Обозначения, вероятно, выглядят немного незнакомыми, но довольно просто понять суть:

```
[ввод] (параметры) { тело }
```

Ввод содержит любые объекты, которые нужны, из области, где определяется лямбда-выражение для выполнения вычислений в теле. *Параметры* определяют имена и типы параметров, вызова с которыми ожидает лямбда-выражение. *Тело* содержит любые вычисления, которые нужно выполнить при вызове. Оно может возвращать или не возвращать значение. Компилятор выведет прототип функции на основе подразумеваемых типов.

В приведенном выше вызове `std::count_if` лямбда-код не нуждался во вводе каких-либо переменных. Вся необходимая информация принимается за один аргумент `number`. Поскольку компилятор знает тип элементов, содержащихся в `x`, вы объявляете тип `number` с помощью `auto`, чтобы компилятор мог определить его самостоятельно. Лямбда-выражение вызывается для каждого элемента `x`, переданного в качестве параметра `number`. В теле лямбда-выражение возвращает `true` только тогда, когда `number` делится на 2, поэтому в `number` включаются только четные числа.

Лямбда-выражений в С нет, и реконструировать их невозможно. Нужно будет объявлять отдельную функцию каждый раз, когда необходим функциональный объект. Невозможно передать объекты в функцию таким же образом.

Обобщенное программирование с помощью шаблонов

Обобщенное программирование — это написание за один раз кода, работающего с разными типами, вместо повторения одного и того же кода, когда копируется и вставляется каждый новый поддерживаемый тип. В языке C++ для создания обобщенного кода используются *шаблоны*. Шаблоны — это особый тип параметра, который указывает компилятору на то, что нужно предоставить широкий диапазон возможных типов.

Вы уже знакомы с шаблонами: все контейнеры stdlib используют их. По большей части тип объектов в этих контейнерах не имеет значения. Например, логика определения количества элементов в контейнере или возврата его первого элемента не зависит от типа элемента.

Предположим, вы хотите написать функцию, которая складывает три числа одного типа. Функция должна принимать любой складываемый тип. В C++ — это простая обобщенная задача программирования, которую можно решить непосредственно с помощью шаблонов (листинг 20).

Листинг 20. Использование шаблонов для создания обобщенной функции сложения

```
template <typename T>
T add(T x, T y, T z) { ❶
  return x + y + z;
}

int main() {
  auto a = add(1, 2, 3);       // a имеет тип int
  auto b = add(1L, 2L, 3L);    // b имеет тип long
  auto c = add(1.F, 2.F, 3.F); // c имеет тип float
}
```

При объявлении `add` ❶ вам не нужно знать `T`. Вам нужно только знать, что все параметры и возвращаемое значение имеют тип `T` и что `T` является слагаемым типом. Когда компилятор обнаруживает вызов `add`, он выводит `T` и генерирует сделанную на заказ функцию от вашего имени. Это серьезное повторное использование кода!

Инварианты классов и управление ресурсами

Возможно, самое большое нововведение C++ в системное программирование — это *жизненный цикл объекта*. Эта концепция берет свое начало в C, где объекты имеют различную продолжительность хранения в зависимости от способа объявления их в коде.

C++ основывается на этой модели управления памятью с помощью конструкторов и деструкторов. Эти специальные функции являются методами, которые принадлежат *пользовательским типам*. Пользовательские типы — это основные строительные блоки приложений C++. Можете представить их как объекты структуры, которые также могут иметь функции.

Конструктор объекта вызывается сразу после начала хранения объекта, а деструктор вызывается непосредственно перед окончанием хранения. И конструктор, и деструктор являются функциями без возвращаемого значения и с тем же именем, что и класс, в котором они содержатся. Чтобы объявить деструктор, добавьте `~` в начало имени класса, как показано в листинге 21.

Первый метод в `Hal` — это *конструктор* ❶. Он создает объект `Hal` и устанавливает его *инварианты класса*. Инварианты — это особенности класса, которые не изменяются после создания. С некоторой помощью компилятора и среды выполнения программист решает, что представляют собой инварианты класса, и обеспечивает их применение в коде. В этом случае конструктор устанавливает инвариант — версию, которая равна 9000. Второй метод — это *деструктор* ❷. `Hal` выводит `"Stop, Dave."`

в консоль всякий раз перед собственным уничтожением. (Заставить `Hal` спеть «Дейзи Белл» — упражнение для читателя в конце главы.)

Листинг 21. Класс Hal, содержащий конструктор и деструктор

```
#include <cstdio>

struct Hal {
  Hal() : version{ 9000 } { // Конструктор ❶
    printf("I'm completely operational.\n");
  }
  ~Hal() { // Деструктор ❷
    printf("Stop, Dave.\n");
  }
  const int version;
};
```

Компилятор обеспечивает автоматический вызов конструктора и деструктора для объектов со статической и локальной длительностью хранения. Для объектов с динамической продолжительностью хранения используются ключевые слова `new` и `delete` на замену `malloc` и `free`, как показано в листинге 22.

Листинг 22. Программа, которая создает и уничтожает объект Hal

```
#include <cstdio>

struct Hal {
--пропуск--
};

int main() {
  auto hal = new Hal{}; // Память выделяется, затем вызывается конструктор
  delete hal;           // Вызывается деструктор, затем память освобождается
}
--------------------------------------------------------------------------------
I'm completely operational.
Stop, Dave.
```

Если по какой-либо причине конструктор не может достичь нормального состояния, то обычно он выдает *исключение*. Как программист на C вы могли иметь дело с исключениями, когда использовались некоторые API операционной системы (например, Windows Structured Exception Handling). Когда генерируется исключение, стек разматывается до тех пор, пока не будет найден обработчик исключения, после чего программа восстанавливается. Разумное использование исключений может привести к очистке кода, потому что проверять наличие ошибок стоит только в тех случаях, когда это имеет смысл. C++ имеет поддержку исключений на уровне языка, как показано в листинге 23.

Можно поместить код, который может вызвать исключение, в блок сразу после `try` ❶. Если в какой-то момент выдается исключение, стек будет разматываться (разрушая любые объекты, выходящие за пределы области видимости) и запускать

любой код, размещенный после выражения `catch` ❷. Если исключение не выдается, код в `catch` никогда не выполняется.

Листинг 23. Блок try-catch

```
#include <exception>

try {
  // Код, который может привести к std::exception ❶
} catch (const std::exception &e) {
  // Восстановление программы в этом месте ❷
}
```

Конструкторы, деструкторы и исключения тесно связаны с другой основной темой C++, связывающей жизненный цикл объекта с ресурсами, которыми он владеет. Это называется концепцией распределения ресурсов при инициализации (RAII) (или *получением конструктора — освобождением деструктора* (*constructor acquires, destructor releases*)). Рассмотрим класс C++ в листинге 24.

Листинг 24. Класс File

```
#include <system error>
#include <cstdio>

struct File {
  File(const char* path, bool write) { ❶
    auto file_mode = write ? "w" : "r"; ❷
    file_pointer = fopen(path, file_mode); ❸
    if (!file_pointer) throw std::system_error(errno, std::system_category()); ❹
  }
  ~File() {
    fclose(file_pointer);
  } FILE* file_pointer;
};
```

Конструктор `File` ❶ принимает два параметра. Первый параметр соответствует пути к файлу, а второй является логическим значением, соответствующим тому, должен ли файл быть открыт для записи (`true`) или чтения (`false`). Значение этого параметра устанавливает `file_mode` ❷ через *тернарный оператор* `? :`. Тернарный оператор вычисляет логическое выражение и возвращает одно из двух значений в зависимости от полученного значения. Например:

```
x ? val_if_true : val_if_false
```

Если логическое выражение `x` истинно, `val_if_true` является значением выражения. Если значение `x` ложно, вместо этого используется значение `val_if_false`.

В фрагменте кода конструктора `File` в листинге 24 конструктор пытается открыть файл в пути `path` с доступом для чтения/записи ❸. Если что-то пойдет не так, при вызове конструктора в `file_pointer` будет передано значение `nullptr`, специальное значение C++, которое похоже на `0`. Если это происходит, генерируется

`system_error` ❹. `system_error` — это просто объект, который инкапсулирует детали системной ошибки. Если `file_pointer` имеет значение, отличное от `nullptr`, его можно использовать. Это инвариант данного класса.

Теперь рассмотрим программу из листинга 25, в которой используется `File`.

Листинг 25. Программа, использующая класс File

```
#include <cstdio>
#include <system error>
#include <cstring>

struct File {
--пропуск--
};

int main() {
  { ❶
    File file("last_message.txt", true); ❷
    const auto message = "We apologize for the inconvenience.";
    fwrite(message, strlen(message), 1, file.file_pointer);
  } ❸
  // last_message.txt закрывается здесь!
  {
    File file("last_message.txt", false); ❹
    char read_message[37]{};
    fread(read_message, sizeof(read_message), 1, file.file_pointer);
    printf("Read last message: %s\n", read_message);
  }
}
--------------------------------------------------------------------------------
We apologize for the inconvenience.
```

Фигурные скобки ❶ ❸ определяют область видимости. Поскольку первый `file` находится в этой области, область видимости определяет время жизни `file`. Как только конструктор вернет ❷, вы будет знать, что `file.file_pointer` является допустимым благодаря инварианту класса; основываясь на дизайне конструктора `File`, известно, что `file.file_pointer` должен быть действителен в течение всего времени существования объекта `File`. С помощью `fwrite` создается сообщение. Нет необходимости явно вызывать `fclose`, потому что срок действия `file` истекает и деструктор очищает `file.file_pointer` самостоятельно ❸. `file` открывается снова, но на этот раз для чтения ❹. Как только конструктор вернет значение, вы будете знать, что *last_message.txt* был успешно открыт и, значит, можно продолжить чтение в `read_message`. После вывода сообщения вызывается деструктор `file` и `file.file_pointer` снова очищается.

Иногда необходимо иметь гибкость динамического выделения памяти одновременно с преимуществами жизненного цикла объекта C++. Это гарантирует, что память не будет потеряна или случайно «использована после освобождения». Именно в этом заключается роль *умных указателей*, которые управляют жизненным циклом дина-

мических объектов через модель владения. Как только умный указатель перестает владеть динамическим объектом, объект разрушается.

Одним из таких умных указателей является `unique_ptr`, который моделирует исключительное владение. Листинг 26 показывает его основное применение.

Листинг 26. Программа, использующая unique_ptr

```
#include <memory>

struct Foundation{
  const char* founder;
};

int main() {
  std::unique_ptr second_foundation{ new Foundation{} }; ❶
  // Получите доступ к переменной-члену founder подобно указателю:
  second_foundation->founder = "Wanda";
} ❷
```

Память динамически выделяется для `Foundation`, и результирующий указатель `Foundation*` передается в конструктор `second_foundation` с помощью скобок ❶. `second_foundation` имеет тип `unique_ptr`, который является просто объектом RAII, оберткой для динамического `Foundation`. Когда `second_foundation` уничтожается ❷, динамический `Foundation` уничтожается соответствующим образом.

Умные указатели отличаются от *обычных*, простых указателей, потому что простой указатель является просто адресом памяти. Вся система управления памятью, связанная с адресом, должна быть организована вручную. С другой стороны, умные указатели обрабатывают все эти беспорядочные детали. Оборачивая динамический объект умным указателем, вы можете быть уверены, что память будет очищена надлежащим образом, как только объект станет ненужным. Компилятор знает, что объект больше не нужен, потому что деструктор умного указателя вызывается при его выходе из области видимости.

Семантика перемещения

Иногда может потребоваться передача права владения объектом, например при использовании `unique_ptr`. Нельзя скопировать `unique_ptr`, поскольку после уничтожения одной из копий `unique_ptr` оставшаяся часть `unique_ptr` будет содержать ссылку на удаленный объект. Вместо копирования объекта используется семантика перемещения в C++ для передачи владения от одного уникального указателя к другому, как показано в листинге 27.

Как и прежде, создается `unique_ptr<Foundation>` ❶. Он используется в течение некоторого времени, а затем нужно будет передать право владения объекту `Mutant`. Функция `move` сообщает компилятору, что вы хотите совершить передачу владения.

После создания `the_mule` ❷ время жизни `Foundation` связывается со временем жизни `the_mule` через переменную-член.

Листинг 27. Программа, перемещающая unique_ptr

```
#include <memory>

struct Foundation{
  const char* founder;
};

struct Mutant {
  // Конструктор устанавливает foundation соответствующим образом:
  Mutant(std::unique_ptr foundation)
    : foundation(std::move(foundation)) {}
  std::unique_ptr foundation;
};

int main() {
  std::unique_ptr second_foundation{ new Foundation{} }; ❶
  // ... используем second_foundation
  Mutant the_mule{ std::move(second_foundation) }; ❷
  // находится в 'перемещенном' состоянии
  // the_mule владеет классом Foundation
}
```

Расслабьтесь и получайте удовольствие

C++ является *основным* языком системного программирования. Большая часть ваших знаний по C будет применяться в C++, но вы также узнаете много новых концепций. Вы можете постепенно включать C++ в свои программы на C, используя Super C. По мере накопления знаний в некоторых более глубоких темах C++ вы обнаружите, что написание современного кода на C++ дает много преимуществ по сравнению с яызком C. Вы сможете емко выражать идеи в коде, используя впечатляющую библиотеку stdlib, работать на более высоком уровне абстракции, использовать шаблоны для повышения производительности во время выполнения и повторного использования кода, а также опираться на жизненный цикл объекта C++ для управления ресурсами.

Я уверен, что инвестиции в изучение C++ принесут огромные дивиденды. А прочитав эту книгу, думаю, вы согласитесь со мной.

ЧАСТЬ 1

Основы языка C++

Сначала мы просто ползаем.
А потом ползаем по битому стеклу.

Скотт Мейерс, «Эффективный STL»

Часть 1 посвящена важнейшим основным концепциям языка C++. Глава 1 расскажет, как настроить рабочую среду и запустить некоторые языковые конструкции, включая основы объектов, базовую абстракцию, использующуюся для программирования на C++.

В следующих пяти главах рассматриваются объекты и типы — сердце и душа C++. В отличие от некоторых других книг по программированию, вам не придется с нуля строить веб-серверы или запускать ракетные корабли. Все программы в первой части просто печатаются в командной строке. Основное внимание уделяется построению ментальной модели языка.

В главе 2 подробно рассматриваются типы — языковая конструкция, определяющая объекты.

Глава 3 на базе главы 2 рассматривает концецию ссылочных типов, которые описывают объекты, ссылающиеся на другие объекты.

Глава 4 описывает жизненный цикл объекта, один из самых мощных аспектов C++.

В главах 5 и 6 рассматриваются полиморфизм во время компиляции с шаблонами и полиморфизм во время выполнения с интерфейсами, которые позволяют писать слабосвязанный и многократно используемый код.

Вооружившись основами объектной модели C++, вы будете готовы погрузиться в главы с 7 по 9. В этих главах представлены выражения, операторы и функции, которые используются для выполнения работы в языке. Может показаться странным, что эти языковые конструкции появляются в конце первой части, но без глубокого знания объектов и их жизненных циклов, не считая самых основных особенностей этих языковых конструкций, их было бы невозможно понять.

Как всеобъемлющий, амбициозный и мощный язык C++ может ошеломить новичка. Чтобы этого не произошло, повествование части 1 ведется последовательно и связно, и ее можно читать как увлекательную историю.

Часть 1 — вступительный взнос. Проделав эту тяжелую работу по изучению основ языка C++ , вы получаете пропуск к «шведскому столу» библиотек и фреймворков, которые можно опробовать во второй части.

1 Создаем и запускаем

...я с такой силой грохнулся вниз, на нашу родную землю, что был совсем оглушен. Благодаря тяжести моего тела, летевшего с такой высоты, я пробил в земле яму глубиной по меньшей мере в девять сажен.

Рудольф Распе,
«Удивительные приключения барона Мюнхгаузена»

Знакомство с языком C++ вы начнете с настройки *среды разработки*, которая представляет собой набор инструментов, позволяющих разрабатывать программное обеспечение на C++. Вы будете использовать среду разработки для компиляции своего первого *консольного приложения* C++, программы, которую можно запустить из командной строки.

Затем вы изучите основные компоненты среды разработки, а также роль, которую они играют в создании вашего первого приложения. В следующих главах будет описано достаточно основ C++ для создания полезных примеров программ.

Язык C ++ имеет репутацию трудного в изучении. Это правда: C++ — большой, сложный и амбициозный язык, и даже опытные программисты на C++ регулярно изучают новые шаблоны, функции и способы использования.

Главная причина в том, что в C++существуют весьма тесные связи. К сожалению, это часто вызывает некоторое беспокойство у новичков. Поскольку концепции C++ тесно связаны между собой, просто непонятно, к чему стремиться. В первой части этой книги я помогу вам пройти через тернии C++, но наш путь должен с чего-то начинаться. Эта глава охватывает достаточно концепций для старта. Пока не углубляйтесь в деатали!

Структура базовой программы на C++

В этом разделе напишем простую программу на C++, а затем скомпилируем и запустим ее. Вы превратите исходный код C++ в удобочитаемые текстовые файлы, называемые *исходными файлами*. Затем используете компилятор для преобразования кода на C++ в исполняемый машинный код, который может быть запущен компьютером.

Начнем погружение и создадим первый исходный файл на C++.

Создание первого исходного файла на C++

Откройте свой любимый текстовый редактор. Если у вас еще нет любимых, попробуйте Vim, Emacs или gedit в Linux; TextEdit на macOS или Блокнот (Notepad) в Windows. Введите код из листинга 1.1 и сохраните полученный файл на рабочий стол под именем *main.cpp*.

Листинг 1.1. Ваша первая программа на C++ выводит Hello, world! на экран

```
#include <cstdio> ❶

int main❷(){
  printf("Hello, world!"); ❸
  return 0; ❹
}
--------------------------------------------------------------------------------
Hello, world! ❸
```

Исходный файл листинга 1.1 компилируется в программу, выводящую символы `Hello, world!` на экран. По соглашению исходные файлы C++ имеют расширение *.cpp*.

ПРИМЕЧАНИЕ

В этой книге в листингах любые выходные данные программы будут добавлены сразу после исходного кода программы; вывод будет отображаться серым цветом. Числовые аннотации будут соответствовать строке кода, на которой произошел вывод. Например, оператор `printf` в листинге 1.1 отвечает за вывод `Hello, world!`, поэтому они используют одну и ту же аннотацию ❸.

Метод main: точка запуска программы

Как показано в листинге 1.1, программы на C++ имеют единственную точку входа, называемую функцией `main` ❷. *Точка входа* — это функция, которая выполняется, когда пользователь запускает программу. *Функции* — это блоки кода, которые могут принимать входные данные, выполнять некоторые инструкции и возвращать результаты.

В `main` вы вызываете функцию `printf`, которая выводит символы `Hello, world!` в консоли ❸. Затем программа завершается, возвращая код выхода 0 в операционную систему ❹. *Коды выхода* — это целочисленные значения, которые операционная система использует для определения того, насколько хорошо работает программа. Как правило, код завершения «ноль» (0) означает, что программа успешно запущена. Другие коды выхода могут указывать на проблему. Наличие оператора возврата в `main` не обязательно; код выхода по умолчанию равен 0.

Функция `printf` не определена в программе; она взята из библиотеки `cstdio` ❶.

Библиотеки: добавление внешнего кода

Библиотеки — это полезные наборы кода, которые можно импортировать в программы, чтобы избежать необходимости изобретать велосипед. Практически каждый язык программирования имеет какой-либо способ включения библиотечных функций в программу:

- В Python, Go и Java — `import`.
- В Rust, PHP и C# — `use/using`.
- В JavaScript, Lua, R и Perl — `require/requires`.
- В C и C++ — `#include`.

В листинге 1.1 использована `cstdio` ❶, библиотека, которая выполняет операции ввода/вывода, такие как вывод в консоль.

Цепочка инструментов компилятора

После написания исходного кода для программы на C++ следующий шаг — превратить исходный код в исполняемую программу. *Цепочка инструментов компилятора* (или *цепочка инструментов*) представляет собой набор из трех элементов, которые запускаются один за другим для преобразования исходного кода в программу.

1. **Препроцессор** выполняет базовые операции с исходным кодом. Например, `#include<cstdio>` ❶ — это директива, которая дает препроцессору команду включить информацию о библиотеке `cstdio` непосредственно в исходный код программы. Когда препроцессор заканчивает обработку исходного файла, он генерирует один блок перевода. Каждая единица трансляции затем передается компилятору для дальнейшей обработки.
2. **Компилятор** читает единицу трансляции и генерирует *объектный файл*. Объектные файлы содержат промежуточный формат, называемый объектным кодом. Эти файлы содержат данные и инструкции в промежуточном формате, который большинство людей не поймут. Компиляторы одновременно работают с одной единицей трансляции, поэтому каждая единица трансляции соответствует одному объектному файлу.

3. **Редактор связей** генерирует программы из объектных файлов. Редакторы связей также несут ответственность за поиск библиотек, включенных в исходный код. Например, при компиляции листинга 1.1 редактор связей найдет библиотеку `cstdio` и добавит все, что нужно программе для использования функции `printf`. Обратите внимание, что заголовок `cstdio` отличается от библиотеки `cstdio`. Заголовок содержит информацию о том, как использовать библиотеку. Вы узнаете больше о библиотеках и организации исходного кода в главе 21.

Настройка среды разработки

Все среды разработки C++имеют способ редактирования исходного кода и цепочку инструментов компилятора для превращения этого исходного кода в программу. Часто среды разработки также содержат *отладчик* — бесценную программу, которая позволяет шаг за шагом проходить программу, чтобы найти ошибки.

Когда все эти инструменты: текстовый редактор, цепочка инструментов компилятора и отладчик — объединены в одну программу, все это в целом называется *интерактивной средой разработки* (interactive development environment, IDE). IDE могут значительно повысить производительность как новичков, так и ветеранов программирования.

ПРИМЕЧАНИЕ

К сожалению, в C++ нет интерпретатора для интерактивного выполнения фрагментов кода C++, в отличие от Python, Ruby и JavaScript, у которых есть интерпретаторы. Существуют некоторые веб-приложения, позволяющие тестировать и делиться небольшими фрагментами кода C++. Посмотрите Wandbox (wandbox.org), который позволяет скомпилировать и запустить код, и компилятор Мэтта Годболта (www.godbolt.org), который позволяет проверять код сборки, сгенерированный вашим кодом. Оба работают с различными компиляторами и системами.

Каждая операционная система имеет свои собственные редакторы исходного кода и цепочку инструментов компилятора, поэтому этот раздел разбит по операционным системам. Перейдите к тому, который имеет отношение к вам.

Windows 10 и выше: Visual Studio

На момент публикации самым популярным компилятором C++ для Microsoft Windows является компилятор Microsoft Visual C++ (MSVC). Самый простой способ получить MSVC — это установить IDE Visual Studio 2017 следующим образом:

1. Загрузите версию Visual Studio 2017 для сообщества. Ссылка доступна по адресу ccc.codes.
2. Запустите установщик, обновите его при необходимости.

3. В окне мастера установки Visual Studio убедитесь, что выбрана опция Desktop Development with C++ Workload.
4. Нажмите Install, чтобы установить Visual Studio 2017 вместе с MSVC.
5. Нажмите Launch для установки Visual Studio 2017. Весь процесс может занять несколько часов в зависимости от скорости вашего компьютера и выбора опций. В среднем установки требуют от 20 до 50 ГБ памяти.

Настройте новый проект:

1. Выберите команду меню File ▸ New ▸ Project.
2. На панели Installed выберите вариант Visual C++ и далее выберите пункт General. Выберите Empty Project на центральной панели.
3. Введите **`hello`** в качестве названия вашего проекта. Открытое окно должно выглядеть так, как показано на рис. 1.1, но расположение может отличаться в зависимости от вашего имени пользователя. Нажмите кнопку OK.

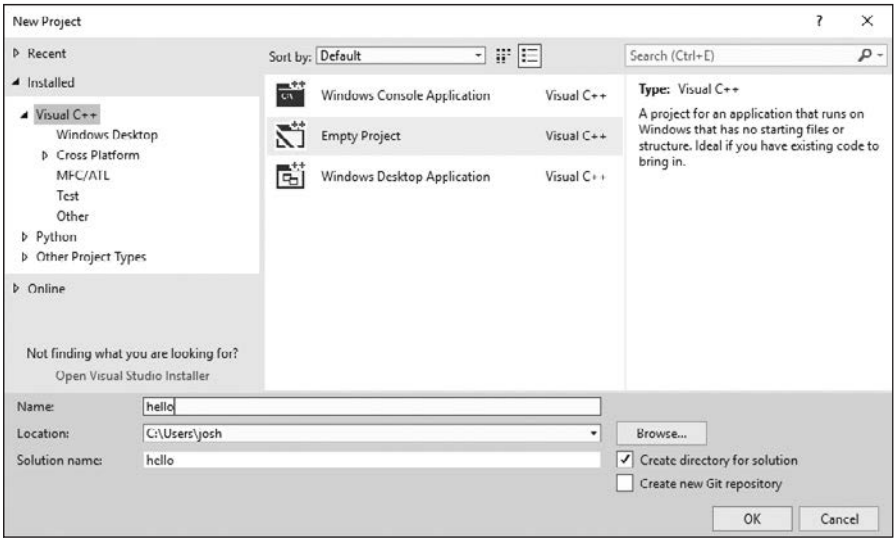


Рис. 1.1. Мастер создания нового проекта Visual Studio 2017

4. На панели Solution Explorer в левой части рабочей области щелкните правой кнопкой мыши по папке Source Files и выберите команду Add ▸ Existing Item в контекстном меню (см. рис. 1.2).
5. Выберите файл *main.cpp*, созданный ранее в листинге 1.1. (В качестве альтернативы, если вы еще не создали этот файл, выберите New Item вместо Existing Item. Назовите файл *main.cpp* и введите содержимое из листинга 1.1 в появившемся окне редактора.)

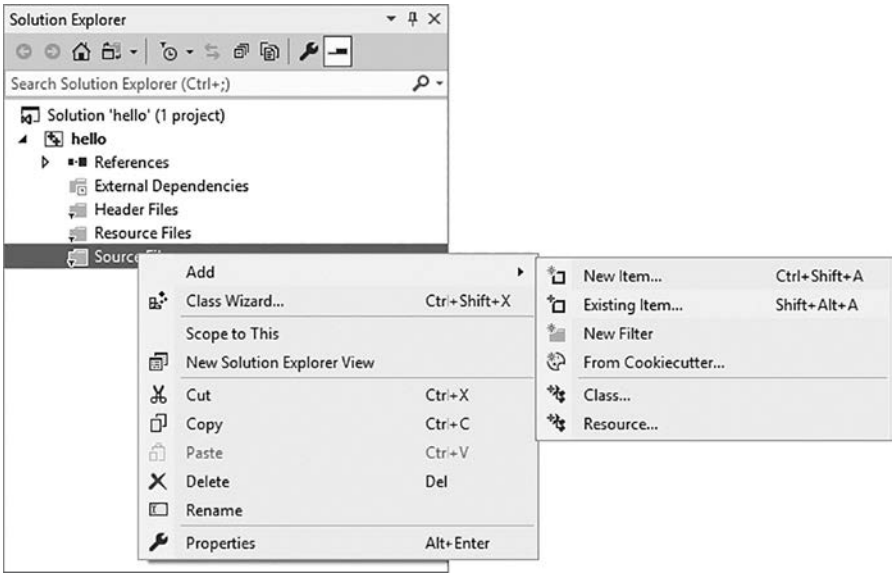


Рис. 1.2. Добавление существующего исходного файла в проект Visual Studio 2017

6. Выберите команду меню Build ▸ Build Solution. Если в окне вывода появятся какие-либо сообщения об ошибках, убедитесь, что вы ввели код из листинга 1.1 правильно. Если вы все еще получаете сообщения об ошибках, внимательно прочитайте их, чтобы найти подсказки.
7. Выберите команду меню Debug ▸ Start Without Debugging или нажмите сочетание клавиш Ctrl+F5, чтобы запустить программу. В консоли должны появиться символы `Hello, world!` (за которыми следует `Press Any Key to Continue`).

macOS: Xcode

При использовании macOS следует установить среду разработки Xcode.

1. Откройте **App Store**.
2. Найдите и установите Xcode IDE. Установка может занять более часа в зависимости от скорости компьютера и подключения к интернету. После завершения установки откройте Terminal и перейдите в каталог, где был сохранен файл *main.cpp*.
3. Введите **`clang ++ main.cpp -o hello`** в Terminal, чтобы скомпилировать программу. Опция `-o` указывает цепочке инструментов, куда записывать вывод. (Если появляются какие-либо ошибки компилятора, проверьте, правильно ли вы ввели программу.)
4. Введите.**`/hello`** в Terminal, чтобы запустить программу. На экране должен появиться текст `Hello, world!`.

Чтобы скомпилировать и запустить программу, откройте IDE Xcode и выполните следующие действия:

1. Выберите команду меню File ▸ New ▸ Project.
2. Выберите команду меню macOS ▸ Command Line Tool и нажмите кнопку Next. В следующем диалоговом окне вы можете изменить расположение каталога файлов проекта. Пока что примите значения по умолчанию и нажмите кнопку Create.
3. Присвойте проекту имя **`hello`** и установите его Type в C++ (см. рис. 1.3).
4. Теперь необходимо импортировать код из листинга 1.1 в проект. Простой способ сделать это — скопировать и вставить содержимое *main.cpp* в *main.cpp* только что созданного проекта. Другой способ — использовать Finder для замены созданного вами *main.cpp* на *main.cpp* проекта. (Обычно вам не придется делать это при создании новых проектов. Это всего лишь побочное действие этой книги, касающейся нескольких операционных сред.)
5. Нажмите кнопку Run.

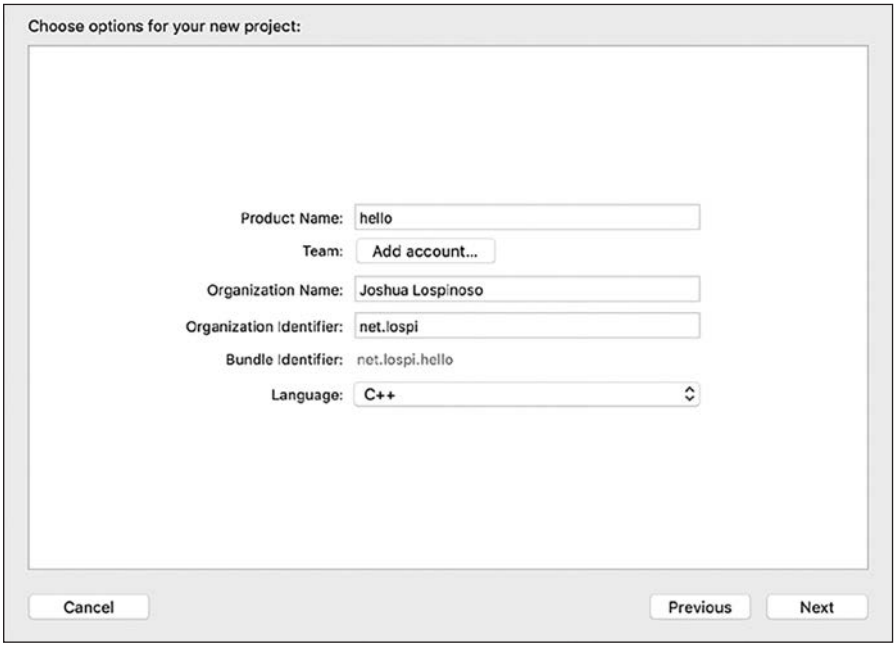


Рис. 1.3. Диалоговое окно нового проекта в Xcode

Linux и GCC

В Linux вы можете выбирать между двумя основными компиляторами C++: GCC и Clang. На момент публикации последняя стабильная версия — 9.1, а последняя

основная версия Clang — 8.0.0. В этом разделе установим оба компилятора. Некоторые пользователи считают сообщения об ошибках одного из них более полезными, чем сообщения другого.

ПРИМЕЧАНИЕ

GCC — это аббревиатура для коллекции компиляторов GNU. В свою очередь, GNU произносится как «га-нью» и является рекурсивной аббревиатурой от «GNU's Not Unix!» (GNU — не Unix!). GNU — это Unix-подобная ОС и набор компьютерных программ.

Попробуйте установить GCC и Clang из диспетчера пакетов операционной системы, но будьте осторожны. В ваших репозиториях по умолчанию могут быть старые версии, которые могут иметь или не иметь поддержку C++17. Если версия не поддерживает C++17, вы не сможете скомпилировать некоторые примеры в книге, поэтому нужно будет установить обновленные версии GCC или Clang. Для краткости в этой главе описывается, как это сделать в Debian и из исходного кода. Вы можете либо изучить, как выполнить соответствующие действия для выбранного вами варианта Linux, либо настроить среду разработки в одной из операционных систем, перечисленных в этой главе.

Установка GCC и Clang в Debian

В зависимости от того, какую ОС содержат Персональные архивы пакетов (Personal Package Archives), когда вы читаете эту главу, вы можете установить GCC 8.1 и Clang 6.0.0 напрямую, используя Advanced Package Tool (APT), менеджер пакетов Debian. В этом разделе показано, как установить GCC и Clang на Ubuntu 18.04, последней версии LTS Ubuntu на момент публикации.

1. Откройте терминал.
2. Обновите установленные на данный момент пакеты:

```
$ sudo apt update && sudo apt upgrade
```

3. Установите GCC 8 и Clang 6.0:

```
$ sudo apt install g++-8 clang-6.0
```

4. Протестируйте GCC и Clang:

```
$ g++-8 --version
g++-8 (Ubuntu 8-20180414-1ubuntu2) 8.0.1 20180414 (experimental) [trunk revision
259383]
Copyright (C) 2018 Free Software Foundation, Inc.
This is free software; see the source for copying conditions. There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
$ clang++-6.0 --version
clang version 6.0.0-1ubuntu2 (tags/RELEASE_600/final)
Target: x86_64-pc-linux-gnu
Thread model: posix
InstalledDir: /usr/bin
```

Если одна из команд возвращает ошибку, в которой говорится, что команда не найдена, соответствующий компилятор не был установлен правильно. Попробуйте найти информацию о полученной ошибке, особенно в документации и на форумах для соответствующего менеджера пакетов.

Установка GCC из источника

Если вы не можете найти последние версии GCC или Clang с установленным на вашей машине менеджером пакетов (или у вашего варианта Unix его нет), вы всегда можете установить GCC из исходного кода. Обратите внимание, что это занимает много времени (до нескольких часов), и вам, возможно, понадобится испачкать руки: при установке часто встречаются ошибки, способ устранения которых нужно найти самостоятельно. Чтобы установить GCC, следуйте инструкциям на сайте gcc.gnu.org. В этом разделе содержится сводка гораздо более обширной документации, доступной на данном сайте.

ПРИМЕЧАНИЕ

Для экономии времени и места здесь не описывается установка Clang. Обратитесь к сайту clang.llvm.org для получения дополнительной информации.

Чтобы установить GCC 8.1 из источника, выполните следующие действия:

1. Откройте терминал.
2. Обновите установленные на данный момент пакеты. Например, при использовании APT нужно выполнить следующую команду:

```
$ sudo apt update && sudo apt upgrade
```

3. Загрузите файлы gcc-8.1.0.tar.gz и gcc-8.1.0.tar.gz.sig с одного из доступных зеркал по адресу gcc.gnu.org/mirrors.html. Эти файлы можно найти в releases/gcc-8.1.0.
4. (Необязательный шаг.) Проверьте целостность пакета. Сначала импортируйте соответствующие ключи GnuPG. Их можно найти в списке на зеркалах сайта. Например:

```
$ gpg --keyserver keyserver.ubuntu.com --recv C3C45C06
gpg: requesting key C3C45C06 from hkp server keyserver.ubuntu.com
gpg: key C3C45C06: public key "Jakub Jelinek " imported
gpg: key C3C45C06: public key "Jakub Jelinek " imported
gpg: no ultimately trusted keys found
gpg: Total number processed: 2
gpg:               imported: 2 (RSA: 1)
```

Проверьте, что вы скачали:

```
$ gpg --verify gcc-8.1.0.tar.gz.sig gcc-8.1.0.tar.gz
gpg: Signature made Wed 02 May 2018 06:41:51 AM DST using DSA key ID C3C45C06
gpg: Good signature from "Jakub Jelinek "
gpg: WARNING: This key is not certified with a trusted signature!
gpg:          There is no indication that the signature belongs to the owner.
Primary key fingerprint: 33C2 35A3 4C46 AA3F FB29 3709 A328 C3A2 C3C4 5C06
```

Выведенные предупреждения означают, что сертификат подписавшего не помечен как проверенный на своем компьютере. Чтобы убедиться, что подпись принадлежит владельцу, необходимо проверить ключ подписи, используя другие средства (например, путем личной встречи с владельцем или проверки внеполосного отпечатка первичного ключа). Для получения дополнительной информации о GNU Privacy Guard (GPG) обратитесь к книге *«PGP & GPG: Email for the Practical Paranoid»* Майкла У. Лукаса или перейдите по адресу gnupg.org/download /integrity_check.html для получения конкретной информации о средствах проверки целостности GPG.

5. Распакуйте пакет (выполнение данной команды может занять несколько минут):

```
$ tar xzf gcc-8.1.0.tar.gz
```

6. Перейдите в только что созданный каталог gcc-8.1.0:

```
$ cd gcc-8.1.0
```

7. Загрузите предварительные требования GCC:

```
$ ./contrib/download_prerequisites
--пропуск--
gmp-6.1.0.tar.bz2: OK
mpfr-3.1.4.tar.bz2: OK
mpc-1.0.3.tar.gz: OK
isl-0.18.tar.bz2: OK
All prerequisites downloaded successfully.
```

8. Настройте GCC с помощью следующих команд:

```
$ mkdir objdir
$ cd objdir
$ ../configure --disable-multilib
checking build system type... x86_64-pc-linux-gnu
checking host system type... x86_64-pc-linux-gnu
--пропуск--
configure: creating ./config.status
config.status: creating Makefile
```

Инструкции доступны по адресу gcc.gnu.org/install/configure.html.

9. Создайте двоичные файлы GCC (при возможности сделайте это за одну ночь, потому что выполнение может занять несколько часов):

```
$ make
```

Полные инструкции доступны по адресу gcc.gnu.org/install/build.html.

10. Проверьте, правильно ли собраны двоичные файлы GCC:

```
$ make -k check
```

Полные инструкции доступны по адресу gcc.gnu.org/install/test.html.

11. Установите GCC:

```
$ make install
```

Эта команда помещает несколько двоичных файлов в исполняемый каталог по умолчанию вашей операционной системы (обычно по адресу /usr/local/bin). Полные инструкции доступны по адресу gcc.gnu.org/install/.

12. Убедитесь, что GCC установлен правильно, введя следующую команду:

```
$ x86_64-pc-linux-gnu-gcc-8.1.0 --version
```

Если вы получите сообщение об ошибке, указывающее, что команда не найдена, установка не удалась. Обратитесь к списку рассылки `gcc-help` по адресу gcc.gnu.org/ml/gcc-help/.

ПРИМЕЧАНИЕ

Возможно, вы захотите связать громоздкий x86_64-pc-linux-gnu-gcc-8.1.0 с чем-то вроде `g++8`, например с помощью такой команды:

```
$ sudo ln -s /usr/local/bin/x86_64-pc-linux-gnu-gcc-8.1.0 /usr/local/bin/g++8
```

13. Перейдите в каталог, где сохранен файл *main.cpp*, и скомпилируйте программу с помощью GCC:

```
$ x86_64-pc-linux-gnu-gcc-8.1.0 main.cpp -o hello
```

14. Флаг `-o` не обязателен; он сообщает компилятору, как назвать результирующую программу. Поскольку вы указали имя программы как `hello`, можно запустить программу, введя **`./hello`**. Если появляются какие-либо ошибки компилятора, убедитесь, что текст программы введен правильно. (Ошибки компилятора должны помочь определить, что что-то пошло не так.)

Текстовые редакторы

Если вы не хотите работать с одной из вышеупомянутых IDE, можно написать код C++ с помощью простого текстового редактора вроде Notepad (Windows), TextEdit (macOS) или Vim (Linux); тем не менее ряд превосходных редакторов разработан специально для программирования на C++. Выберите среду, которая поспособствует вашей максимальной продуктивности.

В Windows или macOS уже установлены высококачественные полнофункциональные IDE, а именно Visual Studio или Xcode. Варианты для Linux включают Qt Creator (www.qt.io/ide/), Eclipse CDT (eclipse.org/cdt/) и CLion JetBrains (www.jetbrains.com/clion/). Если вы являетесь пользователем Vim или Emacs, вы найдете множество плагинов для C++.

ПРИМЕЧАНИЕ

Если важен кроссплатформенный C++, настоятельно рекомендую взглянуть на CLion компании Jetbrains. Хотя CLion является платным продуктом, в отличие от многих своих конкурентов, на момент публикации Jetbrains предлагает бесплатные лицензии для студентов и разработчиков проектов с открытым исходным кодом.

Инициализация C++

Этот раздел даст достаточно контекста для поддержки примеров кода в следующих главах. Подробности можно будет найти там же. Не паникуйте раньше времени!

Система типов C++

C++ — объектно-ориентированный язык. Объекты — это абстракции с состоянием и поведением. Представьте реальный объект, к примеру выключатель света. Вы можете описать его *состояние* как положение, в котором находится переключатель. Он включен или выключен? Какое максимальное напряжение оно может выдержать? В какой комнате дома он находится? Вы также можете описать *поведение* коммутатора. Переключается ли он из одного состояния (включено) в другое (выключено)? Или это регулятор яркости, который может быть настроен на различные состояния между включением и выключением?

Коллекция поведений и состояний, описывающих объект, называется его *типом*. C++ является *строго типизированным языком*, то есть каждый объект имеет предопределенный тип данных.

C++ имеет встроенный целочисленный тип под названием `int`. Объект `int` может хранить целые числа (состояние) и поддерживает множество математических операций (поведение).

Для примера выполнения каких-либо значимых задач с типами `int` создадим несколько объектов `int` и дадим им имена. Именованные объекты называются *переменными*.

Объявление переменных

Переменные объявляются путем предоставления типа, имени и точки с запятой. В следующем примере объявляется переменная `the_answer` с типом `int`:

```
int❶ the_answer❷;
```

За типом `int` ❶ следует имя переменной `the_answer` ❷.

Инициализация состояния переменной

Переменные инициализируются при объявлении. *Инициализация объекта* устанавливает начальное состояние объекта, например его значение. Мы углубимся в детали инициализации в главе 2. Сейчас можно использовать знак равенства (`=`) после объявления переменной, чтобы установить начальное значение переменной. Например, можно объявить и назначить значения `the_answer` одной строкой:

```
int the_answer = 42;
```

После запуска этой строки кода вы получите переменную `the_answer` с типом `int` и значением 42. Переменной можно назначить значение, равное результату вычисления математических выражений, например:

```
int lucky_number = the_answer / 6;
```

Эта строка вычисляет выражение `the_answer/6` и присваивает результат переменной `lucky_number`. Тип `int` поддерживает множество других операций, таких как сложение `+`, вычитание `-`, умножение * и деление по модулю `%`.

ПРИМЕЧАНИЕ

Вы незнакомы с делением по модулю или вам интересно, что происходит при делении двух целых чисел с остатком? Это замечательные вопросы! И подробный ответ на них будет дан в главе 7.

Условные выражения

Условные выражения позволяют принимать решения в программах. Эти решения основаны на логических выражениях, которые оцениваются как `true` или `false`. Например, можно использовать *операторы сравнения*, такие как «больше» или «не равно», для построения логических выражений.

Некоторые основные операторы сравнения, которые работают с типами `int`, приведены в программе в листинге 1.2.

Листинг 1.2. Программа, использующая операторы сравнения

```
int main() {
  int x = 0;
  42 == x; // Равенство
  42 != x; // Неравенство
  100 > x; // Больше
  123 >= x; // Больше или равно
  -10 < x; // Меньше
  -99 <= x; // Меньше или равно
}
```

Эта программа ничего не возвращает (скомпилируйте и запустите листинг 1.2, чтобы убедиться в этом). Хотя программа не производит никаких выходных данных, ее компиляция помогает убедиться, что код на C++ написан правильно. Для создания более интересных программ нужно использовать условный оператор, например `if`.

Оператор `if` содержит логическое выражение и один или несколько вложенных операторов. В зависимости от того, принимает ли логическое значение `true` или `false`, программа решает, какой вложенный оператор выполнить. Существует несколько форм операторов `if`, но в основном они используются следующим образом:

```
if (❶логическое-выражение) ❷оператор
```

Если *логическое-выражение* ❶ истинно, вложенный *оператор* ❷ выполняется; в противном случае — не выполняется.

Иногда нужно, чтобы вместо одного выполнялась группа операторов. Такая группа называется *составным оператором*. Чтобы объявить составной оператор, просто заключите группу операторов в фигурные скобки `{}`. Можно использовать составные операторы внутри операторов `if` следующим образом:

```
if (❶логическое-выражение) { ❷
  оператор1;
  оператор2;
  --пропуск--
}
```

Если *логическое-выражение* ❶ истинно, все операторы в составном *операторе* ❷ выполняются; в противном случае ни один из них не выполняется.

Можно расширить оператор `if`, используя операторы `if` и `else`. Эти необязательные дополнения позволяют описать более сложное поведение ветвления, как показано в листинге 1.3.

Листинг 1.3. Оператор if с другими ветвями if и else

```
❶ if (логическое-выражение-1) оператор-1
❷ else if (логическое-выражение-2) оператор-2
❸ else оператор-3
```

Сначала вычисляется *логическое-выражение-1* ❶. Если *логическое-выражение-1* равно `true`, *оператор-1* вычисляется и оператор `if` прекращает выполнение. Если *логическое-выражение-1* равно `false`, вычисляется *логическое-выражение-2* ❷. Если оно истинно, вычисляется *оператор-2*. В противном случае вычисляется *оператор-3* ❸. Обратите внимание, что *оператор-1*, *оператор-2* и *оператор-3* являются взаимоисключающими и вместе они охватывают все возможные результаты оператора `if`. Только один из трех будет вычислен.

Можно добавить любое количество выражений `else if` или исключить их полностью. Как и в случае с первоначальным оператором `if`, логическое выражение для каждого другого оператора `if` вычисляется по порядку. Когда одно из этих логических выражений вычисляется как `true`, вычисление останавливается и выполняется соответствующий оператор. Если ни один `else if` не имеет значения `true`, *оператор-3* оператора `else` всегда выполняется. (Как и в случае с другими операторами `if`, `else` является необязательным.)

Рассмотрим листинг 1.4, в котором используется оператор `if` для определения, какой оператор нужно вывести.

Скомпилируйте программу и запустите ее. Результат выполнения должен быть равен `Zero`. Теперь измените значение `x` ❶. Что программа выводит сейчас?

ПРИМЕЧАНИЕ

Обратите внимание, что в `main` в листинге 1.4 отсутствует оператор `return`. Поскольку `main` — это специальная функция, операторы `return` являются необязательными.

Листинг 1.4. Программа с условным поведением

```
#include <stdio>

int main() {
  int x = 0; ❶
  if (x > 0) printf("Positive.");
  else if (x < 0) printf("Negative.");
  else printf("Zero.");
}
--------------------------------------------------------------------------------
Zero.
```

Функции

Функции — это блоки кода, которые принимают любое количество входных объектов, которые называются *параметрами* или *аргументами* и могут возвращать выходные объекты функциям, вызвавшим их.

Функции должны быть объявлены в соответствии с общим синтаксисом, показанным в листинге 1.5.

Листинг 1.5. Общий синтаксис для функции C++

```
возвращаемый-тип❶ имя-функции❷ (тип-параметра1 имя-параметра1❸, тип-параметра2
имя-параметра2❹) {
  --пропуск--
  return❺ возвращаемое-значение;
}
```

Первая часть данного объявления функции — это *тип возвращаемой переменной* ❶, такой как `int`. Когда функция возвращает *значение* ❺, тип этого значения должен соответствовать типу возвращаемого значения.

После объявления возвращаемого типа объявляется *имя-функции* ❷. Ряд скобок после имени функции содержит любое количество разделенных запятыми входных параметров, необходимых для функции. Каждый параметр также имеет тип и имя.

Листинг 1.5 имеет два параметра. *Первый параметр* ❸ имеет тип `тип-параметра1` и называется `имя_параметра1`, а *второй параметр* ❹ имеет тип `тип-параметра2` и называется `имя_параметра2`. Параметры представляют объекты, переданные в функцию.

Ряд фигурных скобок, следующих за этим списком, содержит тело функции. Это составной оператор, который содержит логику функции. В рамках этой логики функция может решить вернуть значение функции вызывающему объекту. Функции, которые возвращают значения, будут иметь один или несколько операторов возврата. Как только функция делает возврат, она прекращает выполнение и поток программы возвращается к тому, что вызвало функцию. Давайте посмотрим на примере.

Пример: функция шага

В учебных целях в этом разделе показано, как построить математическую функцию `step_function`, которая возвращает `-1` для всех отрицательных аргументов, `0` для нулевого аргумента и `1` для всех положительных аргументов. Листинг 1.6 показывает, как можно было бы написать функцию `step_function`.

Листинг 1.6. Функция шага, которая возвращает -1 для отрицательных значений, 0 для нуля и 1 для положительных значений

```
int step_function(int ❶x) {
  int result = 0; ❷
  if (x < 0) {
    result = -1; ❸
  } else if (x > 0) {
    result = 1; ❹
  }
return result; ❺
}
```

Функция `step_function` принимает один параметр `x` ❶. Переменная `result` объявляется и инициализируется значением `0` ❷. Затем оператор `if` устанавливает `result` в `-1` ❸, если `x` меньше `0`. Если `x` больше `0`, оператор `if` устанавливает `result` в `1` ❹. Наконец, `result` возвращается вызывающему объекту ❺.

Вызов функций

Чтобы вызвать (или *инициировать*) функцию, нужно использовать имя нужной функции, скобки и список обязательных параметров через запятую. Компилятор читает файлы сверху вниз, поэтому объявление функции должно появляться перед точкой ее первого использования.

Рассмотрим программу в листинге 1.7, которая использует функцию `step_function`.

Листинг 1.7. Программа, использующая функцию `step_function`. (Эта программа не возвращает значение.)

```
int step_function(int x) {
  --пропуск--
}

int main() {
  int value1 = step_function(100); // значение1 равно 1
  int value2 = step_function(0);   // значение2 равно 0
  int value3 = step_function(-10); // значение3 равно -1
}
```

Листинг 1.7 вызывает `step_function` три раза с различными аргументами и присваивает результаты переменным `value1`, `value2` и `value3`.

Было бы неплохо, если бы можно было вывести эти значения. К счастью, существует возможность использовать функцию `printf` для создания выходных данных из

различных переменных. Хитрость заключается в использовании спецификаторов формата `printf`.

Спецификаторы формата printf

В дополнение к выводу постоянных строк (как `Hello, world!` в листинге 1.1) `printf` может объединять несколько значений в красиво отформатированную строку; это особый вид функции, которая может принимать один или несколько аргументов.

Первым аргументом для `printf` всегда является *строка формата*. Строка формата обеспечивает шаблон для строки, которая будет напечатана, и содержит любое количество особых *спецификаторов формата*. Спецификаторы формата сообщают `printf`, как интерпретировать и форматировать аргументы, следующие за строкой формата. Все спецификаторы формата начинаются с `%`.

Например, спецификатор формата для `int` — это `%d`. Всякий раз, когда `printf` видит `%d` в строке формата, он знает, что ожидает аргумент `int` после спецификатора формата. Затем `printf` заменяет спецификатор формата фактическим значением аргумента.

ПРИМЕЧАНИЕ

Функция `printf` является производной от функции `writef`, используемой в BCPL, несуществующем языке программирования, разработанном Мартином Ричардсом в 1967 году. Добавление спецификаторов `%H`, `%I` и `%O` к `writef` приводило к десятичному, шестнадцатеричному и восьмеричному выводу через функции `WRITEHEX`, `WRITED` и `WRITEOCT`. Непонятно, откуда берется спецификатор `%d` (возможно, от `D` в `WRITED`?), но мы это до сих пор не выяснили.

Рассмотрим следующий вызов `printf`, который выводит строки `Ten 10`, `Twenty 20`, `Thirty 30`:

```
printf("Ten %d❶, Twenty %d❷, Thirty %d❸", 10❹, 20❺, 30❻);
```

Первый аргумент `"Ten %d, Twenty %d, Thirty %d"` — это строка формата. Обратите внимание, что существуют три спецификатора формата `%d` ❶ ❷ ❸. Помимо этого, после строки формата в функцию передаются еще три параметра ❹ ❺ ❻. Когда `printf` создает выходные данные, она заменяет аргумент ❶ на аргумент ❹, аргумент в ❷ на аргумент ❺, а аргумент в ❸ на аргумент ❻.

Пересмотр step_function

Давайте посмотрим на другой пример, который использует `step_function`. Листинг 1.8 включает в себя объявления переменных, вызовы функций и спецификаторы формата `printf`.

Листинг 1.8. Программа, которая печатает результаты применения step_function к нескольким целым числам

```
#include <cstdio> ❶

int step_function(int x) { ❷
  --пропуск--
}

int main() { ❸
    int num1 = 42; ❹
    int result1 = step_function(num1); ❺

    int num2 = 0;
    int result2 = step_function(num2);

    int num3 = -32767;
    int result3 = step_function(num3);

    printf("Num1: %d, Step: %d\n", num1, result1); ❻
    printf("Num2: %d, Step: %d\n", num2, result2);
    printf("Num3: %d, Step: %d\n", num3, result3);

    return 0;
}
--------------------------------------------------------------------------------
Num1: 42, Step: 1 ❻
Num2: 0, Step: 0
Num3: -32767, Step: -1
```

Поскольку программа использует `printf`, включена библиотека `cstdio` ❶. `step_function` ❷ определена и ее можно использовать позже в программе, а `main` ❸ устанавливает определенную точку входа.

ПРИМЕЧАНИЕ

Некоторые листинги в этой книге будут основываться друг на друге. В целях экономии места (и спасения деревьев, разумеется) для обозначения отсутствия изменений в повторно используемой части используется нотация `--пропуск--`.

Внутри `main` инициализируются несколько переменных типа `int`, например `num1` ❹. Затем эти переменные передаются в `step_function` и инициализируются переменные результата для хранения возвращаемых значений, например `result1` ❺.

Наконец, с помощью вызовов `printf` выводятся возвращаемые значения. Каждый вызов начинается со строки формата, например `"Num1:%d, Step:%d\n"` ❻. В каждой строке формата есть два спецификатора формата `%d`. В соответствии с требованиями `printf` после строки формата должны быть добавлены два параметра, `num1` и `result1`, которые соответствуют этим двум спецификаторам формата.

ТРУДНОСТИ ОБУЧЕНИЯ: IOSTREAM, PRINTF И ВВОД-ВЫВОД

Мнения о том, какому стандартному методу вывода обучать новичков в C++, сильно разнятся. Одним из вариантов является `printf`, который имеет происхождение, восходящее к C. Другой вариант — `cout`, который является частью библиотеки `iostream`, стандартной библиотеки C++. Эта книга учит так: `printf` появляется в первой части, а `cout` — во второй. И вот почему.

Эта книга дает знания о C++ по кирпичику. Каждая глава разработана последовательно, поэтому не нужно ломать голову, чтобы понять примеры кода. Вы будете точно знать, что делает каждая строка. Поскольку `printf` довольно примитивен, к главе 3 вы накопите достаточно знаний, чтобы четко понимать, как он работает.

В отличие от этого, `cout` включает в себя множество концепций C++, и у вас пока еще не будет достаточного опыта, чтобы понять, как он работает, до конца первой части. (Что такое буфер потока? Что такое оператор `<<`? Что такое метод? Как работает `flush()`? Подождите, `cout` автоматически сбрасывается в деструкторе? Что такое деструктор? Что такое `setf`? И вообще, что такое флаг формата? `BitmaskType`? О боже, что такое манипулятор? И так далее.)

Конечно, у `printf` есть собственные проблемы, и как только вы разберетесь с `cout`, стоит предпочесть его использование. С помощью `printf` можно с легкостью внести несоответствия между спецификаторами формата и аргументами, что может привести к странному поведению, сбоям программы и даже уязвимостям безопасности. Использование `cout` означает, что строки формата больше не требуются, поэтому не нужно запоминать спецификаторы формата. Несоответствие между строками формата и аргументами больше не возникнет. Потоковый ввод-вывод также расширяем, а это означает, что можно интегрировать функции ввода и вывода в пользовательские типы.

В книге преподносится современный C++, но в этой конкретной теме она ставит под угрозу некоторую модернистскую догму в обмен на целенаправленный линейный подход. В качестве преимущества вы будете готовы столкнуться со спецификаторами `printf`, что может произойти в какой-то момент вашей карьеры программиста. Большинство языков, таких как C, Python, Java и Ruby, имеют средства для поддержки спецификаторов `printf`, а в C#, JavaScript и других языках существуют аналоги для этой функции.

Комментарии

Комментарии — это удобочитаемые аннотации, которые можно поместить в исходный код. Вы можете добавлять комментарии к своему коду, используя обозначения `//` или `/**/`. Символ `//` сообщает компилятору игнорировать все, начиная с первой косой черты и заканчивая следующей новой строкой, что означает, что комментарии можно размещать как в фрагментах кода, так и в их собственных строках:

```
// Это комментарий на отдельной строке
int the_answer = 42; // Это встроенный комментарий
```

Нотацию `/**/` можно использовать для включения в код многострочных комментариев:

```
/*
 * Это комментарий
 * Он многострочный
 * Не забудьте закрывающие символы
 */
```

Комментарий начинается с `/*` и заканчивается `*/`. (Звездочки между начальной и конечной косой чертой являются необязательными, но обычно используются.)

Когда использовать комментарии — вопрос вечных споров. Некоторые светила программирования предполагают, что код должен быть настолько выразительным и самообъяснимым, что в комментариях по большей части не было бы необходимости. Они могут говорить, что описательные имена переменных, короткие функции и хорошие тесты — это и есть вся нужная документация. Другие программисты любят ставить комментарии повсюду.

Вы можете продумать свою собственную систему. Компилятор будет полностью игнорировать все, что вы делаете, потому что никогда не интерпретирует комментарии.

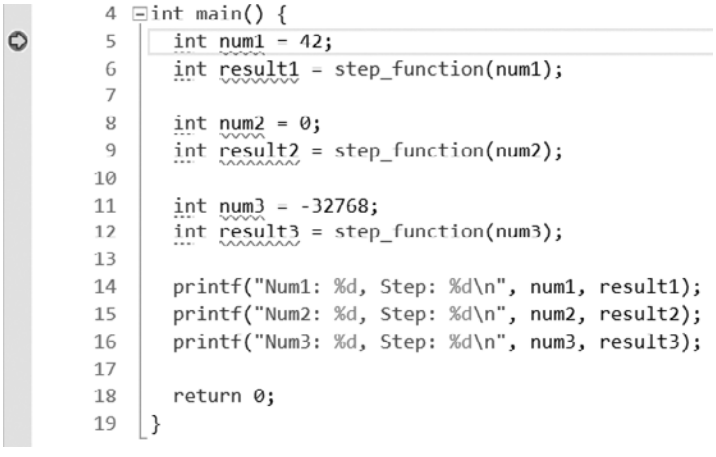
Отладка

Одним из наиболее важных навыков для разработчика является эффективная отладка. Большинство сред разработки имеют инструменты отладки. В Windows, macOS и Linux инструменты отладки превосходны. Научиться правильно их использовать — это инвестиции, которые очень быстро окупаются. В этом разделе представлен краткий обзор того, как использовать отладчик для пошагового выполнения программы в листинге 1.8. Можете сразу перейти к любой наиболее подходящей среде.

Visual Studio

Visual Studio имеет отличный встроенный отладчик. Я предлагаю отлаживать программы в конфигурации *Debug*. Это заставляет цепочку инструментов создавать цель, которая улучшает опыт отладки. Единственная причина отладки в режиме *Release* — диагностика некоторых редких состояний, возникающих в режиме Release, но не в режиме Debug.

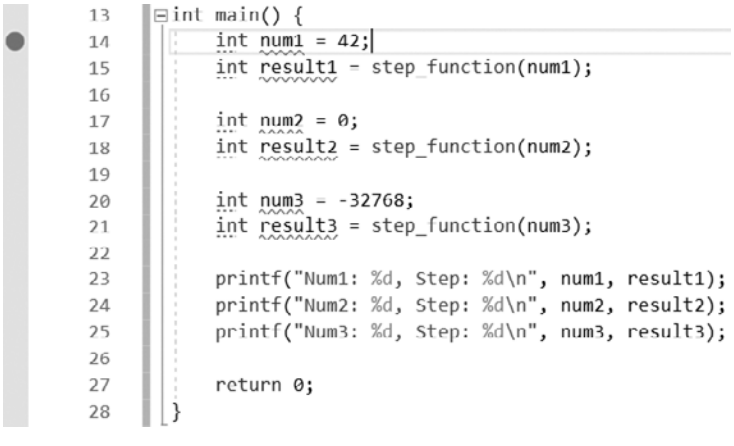
1. Откройте файл *main.cpp* и найдите первую строку `main`.
2. Щелкните по полю слева от номера строки, соответствующего первой строке `main`, чтобы добавить точку останова. Там, где вы щелкнули кнопкой мыши, появится красный кружок, как показано на рис. 1.4.



```
4  int main() {
5      int num1 = 42;
6      int result1 = step_function(num1);
7
8      int num2 = 0;
9      int result2 = step_function(num2);
10
11     int num3 = -32768;
12     int result3 = step_function(num3);
13
14     printf("Num1: %d, Step: %d\n", num1, result1);
15     printf("Num2: %d, Step: %d\n", num2, result2);
16     printf("Num3: %d, Step: %d\n", num3, result3);
17
18     return 0;
19 }
```

Рис. 1.4. Вставка точки останова

3. Выберите команду меню Debug ▸ Start Debugging. Программа выполнится до строчки, где была добавлена точка останова. Отладчик остановит выполнение программы, и появится желтая стрелка, указывающая следующую команду, которая будет запущена, как показано на рис. 1.5.



```
13 int main() {
14     int num1 = 42;
15     int result1 = step_function(num1);
16
17     int num2 = 0;
18     int result2 = step_function(num2);
19
20     int num3 = -32768;
21     int result3 = step_function(num3);
22
23     printf("Num1: %d, Step: %d\n", num1, result1);
24     printf("Num2: %d, Step: %d\n", num2, result2);
25     printf("Num3: %d, Step: %d\n", num3, result3);
26
27     return 0;
28 }
```

Рис. 1.5. Отладчик останавливает выполнение в точке останова

4. Выберите команду меню Debug ▸ Step Over. Операция пошагового выполнения исполняет инструкцию, не входя в какие-либо вызовы функций. По умолчанию горячая клавиша для перехода — F10.
5. Поскольку следующая строка вызывает `step_function`, выберите команду меню Debug ▸ Step Into, чтобы вызвать `step_function` и перейти к ее первой строке. Можно

продолжить отладку этой функции, перейдя по ее инструкциям или пропустив их. По умолчанию горячая клавиша для перехода к следующему шагу — F11.

6. Чтобы разрешить выполнению вернуться к `main`, выберите команду меню Debug ▸ Step Out. По умолчанию сочетание клавиш для этой операции — Shift+F11.
7. Проверьте окно Autos, выбрав команду меню Debug ▸ Windows ▸ Auto. В нем отражаются текущие значения некоторых важных переменных, как показано на рис. 1.6.

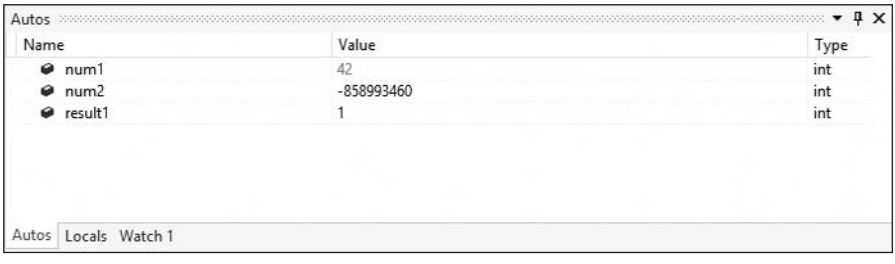


Рис. 1.6. Окно Autos показывает значения переменных в текущей точке останова

Для `num1` установлено значение `42`, а для `result1` — значение `1`. Почему `num2` имеет бредовое значение? Потому что инициализация в `0` еще не произошла: это следующая инструкция для выполнения.

ПРИМЕЧАНИЕ

Отладчик только что подчеркнул очень важную низкоуровневую деталь: выделение памяти для хранения объекта и инициализация его значения — это два разных шага. Вы узнаете больше о распределении памяти и инициализации объектов в главе 4.

Отладчик Visual Studio поддерживает множество других функций. Для получения дополнительной информации просмотрите ссылку на документацию по Visual Studio, доступную по адресу ccc.codes.

Xcode

В Xcode также есть отличный встроенный отладчик, который полностью интегрирован в IDE.

1. Откройте файл *main.cpp* и найдите первую строку `main`.
2. Щелкните первую строку и выберите команду меню Debug ▸ Breakpoints ▸ Add Breakpoint at Current Line. Появляется точка останова, как показано на рис. 1.7.
3. Выберите команду Run. Программа будет выполняться до строки с установленной точкой останова. Отладчик остановит выполнение программы, и появится зеленая стрелка, указывающая следующую команду, которая будет запущена, как показано на рис. 1.8.

```
#include "step_function.h"
#include <cstdio>

int main() {
  int num1 = 42;
  int result1 = step_function(num1);

  int num2 = 0;
  int result2 = step_function(num2);

  int num3 = -32768;
  int result3 = step_function(num3);

  printf("Num1: %d, Step: %d\n", num1, result1);
  printf("Num2: %d, Step: %d\n", num2, result2);
  printf("Num3: %d, Step: %d\n", num3, result3);

  return 0;
}
```

Рис. 1.7. Вставка точки останова

```
#include "step_function.h"
#include <cstdio>

int main() {
  int num1 = 42;                    Thread 1: breakpoint 1.1
  int result1 = step_function(num1);

  int num2 = 0;
  int result2 = step_function(num2);

  int num3 = -32768;
  int result3 = step_function(num3);

  printf("Num1: %d, Step: %d\n", num1, result1);
  printf("Num2: %d, Step: %d\n", num2, result2);
  printf("Num3: %d, Step: %d\n", num3, result3);

  return 0;
}
```

Рис. 1.8. Отладчик останавливает выполнение в точке останова

4. Выберите команду меню Debug ▸ Step Over, чтобы выполнить инструкцию, не вмешиваясь ни в какие вызовы функций. По умолчанию горячая клавиша для этого — F6.
5. Поскольку следующая строка вызывает `step_function`, выберите команду меню Debug ▸ Step Into, чтобы вызвать `step_function` и перескочить на ее первую строку. Можно продолжить отладку этой функции, перейдя по ее инструкциям или пропустив их. По умолчанию горячая клавиша для перехода — F7.

6. Чтобы разрешить выполнению вернуться к `main`, выберите команду меню Debug ▸ Step Out. По умолчанию горячая клавиша для выхода — F8.
7. Проверьте окно Autos в нижней части экрана *main.cpp*. В нем отображены текущие значения некоторых важных переменных, как показано на рис. 1.9.

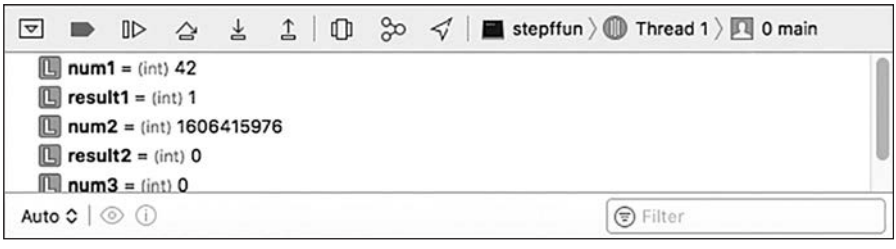


Рис. 1.9. Окно Autos показывает значения переменных в текущей точке останова

Вы можете видеть, что для `num1` установлено значение `42`, а для `result1` — значение `1`. Почему `num2` имеет какое-то странное значение? Потому что инициализация в 0 еще не произошла: это следующая инструкция для выполнения.

Отладчик Xcode поддерживает множество других функций. Для получения дополнительной информации проверьте документацию Xcode по ссылке ccc.codes.

Отладка GCC и Clang с помощью GDB и LLDB

Отладчик проекта GNU (GNU Project Debugger, GDB) — мощный отладчик (www.gnu.org/software/gdb/). С GDB можно взаимодействовать через командную строку. Чтобы включить поддержку отладки во время компиляции с g++ или clang++, добавьте флаг `-g`.

Ваш менеджер пакетов, скорее всего, будет иметь встроенный GDB. Например, чтобы установить GDB с помощью Advanced Package Tool (APT), введите следующую команду:

```
$ sudo apt install gdb
```

Clang также имеет отличный отладчик, называемый отладчиком низкого уровня (LLDB), который можно скачать по адресу lldb.llvm.org/. Он был разработан для работы с командами GDB в этом разделе, поэтому для краткости я не буду явно описывать LLDB. Вы можете отлаживать программы, скомпилированные с поддержкой отладки GCC, используя LLDB, так же как отлаживать программы, скомпилированные с поддержкой отладки Clang, используя GDB.

ПРИМЕЧАНИЕ

Xcode использует LLDB в фоновом режиме.

Чтобы отладить программу в листинге 1.8 с помощью GDB, выполните следующие действия:

1. В командной строке перейдите в папку, где сохранены заголовочные и исходные файлы.
2. Скомпилируйте программу с поддержкой отладки:

```
$ g++-8 main.cpp -o stepfun -g
```

3. Начните отладку программы с использованием `gdb`; в интерактивной консоли должно появиться следующее:

```
$ gdb stepfun
GNU gdb (Ubuntu 7.7.1-0ubuntu5~14.04.2) 7.7.1
Copyright (C) 2014 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying" and
"show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from stepfun...done.
(gdb)
```

4. Для добавления точки останова используйте команду `break`, которая принимает один параметр, соответствующий имени исходного файла и строке, на которой нужно остановиться, через двоеточие (`:`). Например, предположим, что необходимо остановиться на первой строке файла *main.cpp*. В листинге 1.8 это показано в строке 5 (хотя может потребоваться изменить размещение в зависимости от того, как был написан исходный код). Вы можете создать точку останова, используя следующую команду в диалоговом окне (`gdb`):

```
(gdb) break main.cpp:5
```

5. Также можно указать GDB прерывать определенную функцию по имени:

```
(gdb) break main
```

6. В любом случае, теперь программа может быть запущена:

```
(gdb) run
Starting program: /home/josh/stepfun
Breakpoint 1, main () at main.cpp:5
5         int num1 = 42;
(gdb)
```

7. Для прохождения инструкции используйте команду `step`, чтобы последовательно пройти каждую строку программы, включая переход к функциям:

```
(gdb) step
6         int result1 = step_function(num1);
```

8. Чтобы продолжить пошаговое выполнение, нажмите клавишу Enter, тем самым вернувшись к последней команде:

```
(gdb)
step_function (x=42) at step_function.cpp:4
```

9. Чтобы выйти из выполнения функции, используйте команду `finish`:

```
(gdb) finish
Run till exit from #0 step_function (x=42) at step_function.cpp:7
0x0000000000400546 in main () at main.cpp:6
6         int result1 = step_function(num1);
Value returned is $1 = 1
```

10. Чтобы выполнить инструкцию, не переходя к функции, используйте команду `next`:

```
(gdb) next
8         int num2 = 0;
```

11. Для проверки текущего значения переменных используйте команду `info locals`:

```
(gdb) info locals
num2 = -648029488
result2 = 32767
num1 = 42
result1 = 1
num3 = 0
result3 = 0
```

    Обратите внимание, что любые переменные, которые еще не были инициализированы, не будут иметь разумных значений.

12. Чтобы продолжить выполнение до следующей точки останова (или до завершения программы), используйте команду `continue`:

```
(gdb) continue
Continuing.
Num1: 42, Step: 1
Num2: 0, Step: 0
Num3: -32768, Step: -1
[Inferior 1 (process 1322) exited normally]
```

13. Используйте команду `quit` для выхода из `gdb` в любой момент.

GDB поддерживает множество других функций. Для получения дополнительной информации ознакомьтесь с документацией по адресу sourceware.org/gdb/current/onlinedocs/gdb/.

Итоги

В этой главе вы познакомились со средой разработки C++ и скомпилировали свою первую программу на C++. Узнали о компонентах цепочки инструментов сборки и о роли, которую они играют в процессе компиляции. Затем вы изучили несколько важных тем C++, таких как типы, объявления переменных, операторы, условные выражения, функции и `printf`. Глава завершилась руководством по настройке отладчика и пошаговой проработке проекта.

ПРИМЕЧАНИЕ

Если у вас возникли проблемы с настройкой среды, поищите сообщения об ошибках в интернете. Если это не помогло, опубликуйте свой вопрос в Stack Overflow по адресу stackoverflow.com, блоге C++ по адресу www.reddit.com/r/cpp_questions/ или канале C++ в Slack по адресу cpplang.now.sh.

Упражнения

Попробуйте выполнить эти упражнения, чтобы закрепить на практике то, что вы узнали в этой главе. (Дополнительный код книги доступен по адресу ccc.codes.)

1.1. Создайте функцию `absolute_value`, которая возвращает абсолютное значение одного аргумента. Абсолютное значение целого числа `x` является следующим: `x` (само по себе), если `x` больше или равно `0`; в противном случае это `x` в степени `-1`. Можно использовать программу в листинге 1.9 в качестве шаблона.

Листинг 1.9. Шаблон для программы, использующей функцию `absolute_value`

```
#include <cstdio>

int absolute_value(int x) {
  // Ваш код
}

int main() {
  int my_num = -10;
  printf("The absolute value of %d is %d.\n", my_num, absolute_value(my_num));
}
```

1.2. Попробуйте запустить программу с разными значениями. Совпадают ли полученные значения с ожидаемыми?

1.3. Запустите программу с отладчиком, шаг за шагом выполняя каждую инструкцию.

1.4. Напишите другую функцию с именем `sum`, которая принимает два аргумента типа `int` и возвращает их сумму. Как можно изменить шаблон в листинге 1.9, чтобы проверить новую функцию?

1.5. C++ имеет активное онлайн-сообщество, и интернет переполнен отличными материалами, связанными с C++. Изучите подкаст CppCast по адресу cppcast.com. Найдите на YouTube видео CppCon и C++Now. Добавьте cppreference.com и www.cplusplus.com в закладки браузера.

1.6. Наконец, загрузите копию стандарта C++ 17 Международной организации по стандартизации (ISO) со страницы isocpp.org/std/the-standard/. К сожалению, официальный стандарт ISO защищен авторским правом и должен быть приобретен. К счастью, можно бесплатно скачать «черновик», который минимально отличается от официальной версии.

ПРИМЕЧАНИЕ

Поскольку номера страниц стандарта ISO отличаются от версии к версии, эта книга будет ссылаться на отдельные разделы, используя ту же схему именования, что и сам стандарт. Данная схема цитирует разделы, заключая имя раздела в квадратные скобки. Подразделы разделены точками. Например, для цитирования раздела об объектной модели C++ (C++ Object Model), который находится в разделе «Введение» (Introduction), будет указано [intro.object].

Что еще почитать?

- «Программист-прагматик. Путь от подмастерья к мастеру», Эндрю Хант, Дэвид Томас (Лори, 2009)
- «The Art of Debugging with GDB, DDD, and Eclipse», Norman Matloff, Peter Jay Salzman (No Starch Press, 2008)
- «Email for the Practical Paranoid», Michael W. Lucas (Starch Press, 2006)
- «GNU Make Book», John Graham-Cumming (No Starch Press, 2015)

2 Типы

Хардин однажды сказал: «Чтобы преуспеть в делах, планировать мало. Нужно уметь импровизировать». Я буду импровизировать.

Айзек Азимов, «Основание»

Как обсуждалось в главе 1, тип определяет, как объект будет интерпретироваться и использоваться компилятором. Каждый объект в программе на C++ имеет тип. Эта глава начинается с подробного обсуждения основных типов, а затем познакомит читателя с пользовательскими типами. Попутно вы узнаете о нескольких структурах потока управления.

Основные типы

Основные типы являются наиболее базовыми типами объектов и включают в себя целые числа, числа с плавающей точкой, символы, логическое значение и `void`. Кто-то называет фундаментальные типы *примитивными* или *встроенными*, потому что они являются частью основного языка и почти всегда доступны. Эти типы будут работать на любой платформе, но их функции, такие как размер и расположение памяти, зависят от реализации.

Основные типы соблюдают баланс. С одной стороны, они пытаются отобразить прямую связь между конструкцией C++ и компьютерным оборудованием; с другой — они упрощают написание кроссплатформенного кода, позволяя программисту писать код, работающий на многих платформах. В следующих разделах приведены дополнительные сведения об этих основных типах.

Целочисленные типы

Целочисленные типы хранят целые числа: те, что можно записать без дробного компонента. Существуют четыре варианта целочисленных типов: *short int, int, long int* и *long long int*. Каждый из них может быть знаковым или беззнаковым. *Знаковая* переменная может быть положительной, отрицательной или нулевой, а *беззнаковая* переменная должна быть неотрицательной.

Целочисленные типы по умолчанию являются знаковыми и имеют тип `int`, что означает, что вы можете использовать следующие сокращенные записи в своих программах: `short`, `long` и `long long` вместо `short int`, `long int` и `long long int`. В табл. 2.1 перечислены все доступные целочисленные типы C++ независимо от того, знаковые они или нет, размер каждого (в байтах) на разных платформах, а также спецификатор формата для каждого.

Таблица 2.1. Целочисленные типы, их размеры и спецификаторы формата

Тип	Знаковый	Размер в байтах				Спецификатор формата `printf`
		32-битная ОС		64-битная ОС		
		Windows	Linux/macOS	Windows	Linux/macOS	
`short`	Да	2	2	2	2	`%hd`
`unsigned short`	Нет	2	2	2	2	`%hu`
`int`	Да	4	4	4	4	`%d`
`unsigned int`	Нет	4	4	4	4	`%u`
`long`	Да	4	4	4	8	`%ld`
`unsigned long`	Нет	4	4	4	8	`%lu`
`long long`	Да	8	8	8	8	`%lld`
`unsigned long long`	Нет	8	8	8	8	`%llu`

Обратите внимание, что размеры целочисленных типов различаются для разных платформ: 64-битные Windows и Linux/macOS имеют разные размеры для long целых чисел (4 и 8 соответственно).

Обычно компилятор предупреждает о несоответствии между спецификатором формата и целочисленным типом. Но стоит убедиться, что спецификаторы формата верны при использовании их в операторах `printf`. Здесь появляются спецификаторы формата, поэтому в следующих примерах можно будет вывести целые числа в консоль.

Литерал — это жестко закодированное значение в программе. Можно использовать одно из четырех жестко запрограммированных *целочисленных литеральных* представлений:

- **`binary`** — использует префикс `0b`;
- **`octal`** — использует префикс `0`;
- **`decimal`** — это значение по умолчанию;
- **`hexadecimal`** — использует префикс `0x`.

ПРИМЕЧАНИЕ

Если нужно применить гарантированные размеры целых чисел, то можно использовать целочисленные типы из библиотеки `<cstdint>`. Например, если требуется целое число со знаком, равное 8, 16, 32 или 64 битам, вы можете использовать `int8_t`, `int16_t`, `int32_t` или `int64_t`. Здесь найдутся варианты самых быстрых, наименьших, максимальных, знаковых и беззнаковых целочисленных типов, соответствующих вашим требованиям. Но поскольку этот заголовок может не быть доступен на каждой платформе, стоит использовать типы `cstdint` только тогда, когда нет альтернативы.

Это четыре разных способа написания одного и того же набора целых чисел. Например, в листинге 2.1 показано, как можно назначить несколько целочисленных переменных с целочисленными литералами, используя каждое из недесятичных представлений.

Листинг 2.1. Программа, которая определяет несколько целочисленных переменных и выводит их с соответствующим спецификатором формата

```
#include <cstdio>

int main() {
  unsigned short a = 0b10101010; ❶
  printf("%hu\n", a);
  int b = 0123; ❷
  printf("%d\n", b);
  unsigned long long d = 0xFFFFFFFFFFFFFFFF; ❸
  printf("%llu\n", d);
}
--------------------------------------------------------------------------------
170 ❶
83 ❷
18446744073709551615 ❸
```

Данный пример использует каждое из недесятичных целочисленных представлений (двоичное ❶, восьмеричное ❷ и шестнадцатеричное ❸) и выводит каждое с помощью `printf`, используя соответствующий спецификатор формата, указанный в табл. 2.1. Вывод каждого `printf` соответствует комментарию на новой строке.

ПРИМЕЧАНИЕ

Целочисленные литералы могут содержать любое количество одинарных кавычек (') для удобства чтения. Это полностью игнорируется компилятором. Например, 1000000 и 1'000'000 — это целочисленные литералы, равные одному миллиону.

Иногда бывает полезно вывести целое число без знака в его шестнадцатеричном представлении или (изредка) в восьмеричном представлении. Для этих целей можно использовать спецификаторы `printf %x` и `%o` соответственно, как показано в листинге 2.2.

Листинг 2.2. Программа, которая использует восьмеричные и шестнадцатеричные представления целых беззнаковых чисел

```
#include <cstdio>

int main() {
  unsigned int a = 3669732608;
  printf("Yabba %x❶!\n", a);
  unsigned int b = 69;
  printf("There are %u❷,%o❸ leaves here.\n", b❹, b❺);
}
--------------------------------------------------------------------------------
Yabba dabbad00❶!
There are 69❷,105❸ leaves here.
```

Шестнадцатеричное представление десятичного числа `3669732608` — это `dabbad00`, которое появляется в первой строке вывода в результате применения спецификатора шестнадцатеричного формата `%x` ❶. Десятичное число 69 — это 105 в восьмеричной системе счисления. Спецификаторы формата для целого беззнакового числа `%u` ❷ и восьмеричного целого `%o` ❸ соответствуют параметрам в ❹ и ❺ соответственно. Оператор `printf` подставляет эти величины ❷ ❸ в строку формата, получая сообщение `There are 69, 105 leaves in here`.

ВНИМАНИЕ

Восьмеричный префикс является пережитком языка B, который использовали еще во времена компьютера PDP-8 и вездесущих восьмеричных литералов. Язык C и расширение C++ продолжают сомнительную традицию. Следует проявлять осторожность, например, при жестком кодировании почтовых индексов:

```
int mit_zip_code = 02139; // Не скомпилируется
```

Уберите начальные нули на десятичных литералах; в противном случае они перестанут быть десятичными. Эта строка не компилируется, потому что 9 не является восьмеричной цифрой.

По умолчанию тип целочисленного литерала — один из следующих: `int`, `long` или `long long`. Тип целочисленного литерала является наименьшим из подходящих из этих трех типов. (Это определяется языком и будет выполняться компилятором.)

Для большего контроля вы можете применить *суффиксы* к целочисленному литералу, чтобы указать его тип (суффиксы не чувствительны к регистру, поэтому их стиль может быть произвольным):

- суффикс `unsigned` — `u` или `U`
- суффикс `long` — `l` или `L`
- суффикс `long long` –`ll` или `LL`

Можно комбинировать суффикс `unsigned` с суффиксом `long` или `long long`, чтобы указать и знаковость, и размер. В табл. 2.2 представлены возможные типы, которые могут получиться при комбинации суффиксов. Разрешенные типы отмечены галочкой (✓). Для двоичных, восьмеричных и шестнадцатеричных литералов можно опустить суффикс `u` или `U`. Они обозначены звездочкой (*).

Таблица 2.2. Целочисленные суффиксы

Тип	(нет суффикса)	l/L	ll/LL	u/U	ul/UL	ull/ULL
`int`	✓					
`long`	✓	✓				
`long long`	✓	✓	✓			
`unsigned int`	*			✓		
`unsigned long`	*	*		✓	✓	
`unsigned long long`	*	*	*	✓	✓	✓

Результирующий тип — это наименьший допустимый тип, который все еще соответствует целочисленному литералу. Это означает, что из всех типов, разрешенных для конкретного целого числа, будет применяться наименьший тип. Например, целочисленный литерал `112114` может быть `int`, `long` или `long long`. Поскольку `int` может хранить `112114`, результирующим целочисленным литералом является `int`. Если все-таки нужно использовать, скажем, `long`, укажите вместо этого `112114L` (или `1121141`).

Типы с плавающей точкой

Типы с плавающей запятой хранят аппроксимации действительных чисел (которые в нашем случае могут быть определены как любое число с десятичной точкой и дробной частью, например 0,33333 или 98,6). Хотя невозможно точно представить произвольное действительное число в памяти компьютера, можно сохранить приблизительное значение. Если в это трудно поверить, просто представьте число, подобное π, которое имеет бесконечно много цифр. Учитывая конечную компьютерную память, как можно было бы представлять бесконечное множество цифр?

Как все типы, типы с плавающей точкой занимают ограниченный объем памяти, что называется *точностью* типа. Чем выше точность типа с плавающей точкой, тем точнее он будет при приближении к действительному числу. C++ предлагает три уровня точности для приближений:

- **`float`** — одинарная точность
- **`double`** — двойная точность
- **`long double`** — повышенная точность

Как и с целочисленными типами, каждое представление с плавающей точкой зависит от реализации. В этом разделе не будет подробно рассказываться о типах с плавающей точкой, но обратите внимание, что в этих реализациях есть существенный нюанс.

В основных операционных системах для ПК уровень `float` обычно имеет точность 4 байта. Уровни `double` и `long double` обычно имеют 8 байтов точности (*двойная точность*).

Большинство пользователей, не использующих научные вычислительные приложения, могут безопасно игнорировать детали представления с плавающей точкой. В таких случаях хорошим общим правилом является использование `double`.

ПРИМЕЧАНИЕ

Тем, кто не может просто так взять и пропустить подробности, я предлагаю ознакомиться со спецификацией чисел с плавающей точкой, относящейся к конкретной аппаратной платформе. Преимущественная реализация хранения и проведения вычислений с плавающей точкой описана в стандарте IEEE для вычислений с плавающей точкой (The IEEE Standard for FloatingPoint Arithmetic), IEEE 754.

Литералы с плавающей точкой

Литералы с плавающей точкой по умолчанию имеют двойную точность. Если нужна одинарная точность, используйте суффикс `f` или `F`; для повышенной точности используйте `l` или `L`, как показано здесь:

```
float a = 0.1F;
double b = 0.2;
long double c = 0.3L;
```

Также можно использовать научные обозначения в литералах:

```
double plancks_constant = 6.62607004❶e-34❷;
```

Не допускаются пробелы между *значащей* частью числа (основанием ❶) и *суффиксом* (экспоненциальной частью ❷).

Спецификаторы формата с плавающей точкой

Спецификатор формата `%f` отображает число с плавающей запятой с десятичными цифрами, тогда как `%e` отображает то же число в научной нотации. Вы можете позволить `printf` решить, какой из этих двух параметров использовать, применив спецификатор формата `%g`, который выбирает более компактный из `%e` или `%f`.

Для представления `double` просто добавьте `l` (`L` в нижнем регистре) к желаемому спецификатору; для `long double` добавьте `L`. Например, если нужно получить `double` с десятичными цифрами, укажите `%lf`, `%le` или `%lg`; для `long double` стоит указать `%Lf`, `%Le` или `%Lg`.

Рассмотрим листинг 2.3, в котором рассматриваются различные варианты вывода числа с плавающей точкой.

Листинг 2.3. Программа, выводящая несколько чисел с плавающей точкой

```
#include <cstdio>

int main() {
  double an = 6.0221409e23; ❶
  printf("Avogadro's Number: %le❷ %lf❸ %lg❹\n", an, an, an);
  float hp = 9.75; ❺
  printf("Hogwarts' Platform: %e %f %g\n", hp, hp, hp);
}
--------------------------------------------------------------------------------
Avogadro's Number: 6.022141e+23❷ 602214090000000006225920.000000❸
6.02214e+23❹
Hogwarts' Platform: 9.750000e+00 9.750000 9.75
```

Эта программа объявляет `double` под названием `an` ❶. Спецификатор формата `%le` ❷ предоставляет научную запись `6.022141e-23`, а `%lf` ❸ обозначает десятичное представление `602214090000000006225920.000000`. Спецификатор `%lg` ❹ выбрал научную нотацию `6.02214e-23`. `float` под названием `hp` ❺ выдает аналогичные выходные данные в `printf` с использованием спецификаторов `%e` и `%f`. Но спецификатор формата `%g` решил предоставить десятичное представление `9,75`, а не научную запись.

Используйте `%g` для вывода типов с плавающей точкой.

ПРИМЕЧАНИЕ

На практике префикс `l` в спецификаторах формата для `double` можно опустить, потому что `printf` продвигает аргументы `float` с точностью `double`.

Символьные типы

Символьные типы хранят данные о естественном языке. Существует шесть типов символов:

- **`char`** — тип по умолчанию, размером всегда в 1 байт. Может быть знаковым или беззнаковым. (Пример: ASCII.)
- **`char16_t`** — используется для 2-байтовых наборов символов. (Пример: UTF-16.)
- **`char32_t`** — используется для 4-байтовых наборов символов. (Пример: UTF-32.)
- **`signed char`** — то же, что и `char`, но гарантированно знаковый.
- **`unsigned char`** — то же, что и `char`, но гарантированно беззнаковый.
- **`wchar_t`** — достаточно большой, чтобы содержать самый большой символ языкового стандарта реализации. (Пример: Юникод.)

Типы символов `char`, `signed char` и `unsigned char` называются *узкими символами*, тогда как `char16_t`, `char32_t` и `wchar_t` — *широкими символами* из-за их относительных требований к памяти.

Символьные литералы

Символьный литерал — это один постоянный символ. Одинарные кавычки (`''`) окружают все символы. Если символ имеет любой тип, кроме `char`, также необходимо предоставить префикс: `L` для `wchar_t`, `u` для `char16_t` и `U` для `char32_t`. Например, `'J'` объявляет литерал `char`, а `L'J'` объявляет `wchar_t`.

Экранированные последовательности

Некоторые символы не отображаются на экране. Они служат для перемещения курсора в левую часть экрана (возврат каретки) или вниз на одну строку (новая строка). Другие символы могут отображаться на экране, но являются частью синтаксиса языка C++, такие как одинарные или двойные кавычки, поэтому использовать их следует очень осторожно. Чтобы поместить эти символы в `char`, используйте *экранированные последовательности*, перечисленные в табл. 2.3.

Таблица 2.3. Зарезервированные символы и их экранированные последовательности

Значение	Экранированная последовательность
Новая строка	`\n`
Табуляция (горизонтальная)	`\t`
Табуляция (вертикальная)	`\v`
Возврат на одну позицию	`\b`
Возврат каретки	`\r`
Прогон страницы	`\f`
Оповещение	`\a`
Обратная косая	`\\`
Знак вопроса	`?` или `\?`
Одинарная кавычка	`\'`
Двойная кавычка	`\"`
Нулевой символ	`\0`

Экранированные символы Unicode

Можно указать символьные литералы Unicode, используя *универсальные имена символов*, и сформировать универсальное имя символа одним из двух способов: префикс `\u`, за которым следует 4-значный код символа Unicode, или префикс `\U`, за которым следует 8-значный код символа Unicode. Например, можно представить символ `A` как `'\u0041'`, а символ «кружка пива» 🍺 — как `U'\U0001F37A`.

Спецификаторы формата

Спецификатор формата `printf` для `char` — это `%c`. Спецификатор формата `wchar_t` — `%lc`.

Листинг 2.4 инициализирует два символьных литерала, `x` и `y`. Эти переменные используются для создания вызова `printf`.

Листинг 2.4. Программа, которая инициализирует несколько символьных переменных и выводит их

```
#include <cstdio>

int main() {
  char x = 'M';
  wchar_t y = L'Z';
  printf("Windows binaries start with %c%lc.\n", x, y);
}
--------------------------------------------------------------------------------
Windows binaries start with MZ.
```

Эта программа выводит: *«Windows binaries start with MZ»*. Хотя *M* — это узкий символ `char`, а *Z* — широкий, `printf` работает, потому что программа использует правильные спецификаторы формата.

ПРИМЕЧАНИЕ

Первые два байта всех двоичных файлов Windows — это символы M и Z, дань уважения Марку Збиковски, разработчику формата исполняемых двоичных файлов MS-DOS.

Логические типы

Логические (булевы) типы имеют два состояния: `true` (истина) и `false` (ложь). Единственный логический тип — `bool`. Целочисленные типы и типы `bool` преобразуются легко: состояние `true` преобразуется в `1`, а `false` — в `0`. Любое ненулевое целое число преобразуется в `true`, а `0` — в `false`.

Логические литералы

Чтобы инициализировать логические типы, используйте два логических литерала, `true` и `false`.

Спецификаторы формата

Для `bool` не существует спецификатора формата, но можно использовать спецификатор формата `int` `%d` в `printf`, чтобы получить `1` для `true` и `0` для `false`. Причина в том, что `printf` добавляет любое целое значение, меньшее, чем `int`, в `int`. Листинг 2.5 показывает, как объявить логическую переменную и проверить ее значение.

Листинг 2.5. Вывод переменных bool с помощью оператора printf

```
#include <cstdio>

int main() {
  bool b1 = true; ❶ // b1 истинно
  bool b2 = false; ❷ // b2 ложно
  printf("%d %d\n", b1, b2); ❸
}
--------------------------------------------------------------------------------
1 0 ❸
```

`b1` инициализируется со значением `true` ❶, а `b2` — со значением `false` ❷. Выводя `b1` и `b2` как целые числа (с использованием спецификаторов формата `%d`), вы получаете `1` для `b1` и `0` для `b2` ❸.

Операторы сравнения

Операторы — это функции, которые выполняют вычисления над *операндами*. Операнды — это просто объекты (раздел «Логические операторы» охватывает полный список операторов). Чтобы получить содержательные примеры с использованием типов `bool`, кратко рассмотрим операторы сравнения в этом разделе и логические операторы в следующем.

Можно использовать несколько операторов для построения логических выражений. Напомним, что операторы сравнения принимают два аргумента и возвращают логическое значение. Доступные операторы: равенство (`==`), неравенство (`!=`), больше (`>`), меньше (`<`), больше или равно (`>=`) и меньше или равно (`<=`).

Листинг 2.6 показывает, как можно использовать эти операторы для получения логических значений.

Листинг 2.6. Использование операторов сравнения

```
#include <cstdio>

int main() {
  printf(" 7 == 7: %d❶\n", 7 == 7❷);
  printf(" 7 != 7: %d\n", 7 != 7);
  printf("10 > 20: %d\n", 10 > 20);
  printf("10 >= 20: %d\n", 10 >= 20);
  printf("10 < 20: %d\n", 10 < 20);
  printf("20 <= 20: %d\n", 20 <= 20);
}
--------------------------------------------------------------------------------
 7 ==  7: 1 ❶
 7 !=  7: 0
10 >  20: 0
10 >= 20: 0
10 <  20: 1
20 <= 20: 1
```

Каждое сравнение выдает логический результат ❷, а оператор `printf` выводит логическое значение как `int` ❶.

Логические операторы

Логические операторы вычисляют логические выражения для типов `bool`. Операторы характеризуются количеством операндов, которые они принимают. *Унарный оператор* принимает один операнд, *бинарный оператор* — два, *тернарный оператор* — три и т. д. Операторы классифицируются далее по описанию типов их операндов.

Унарный оператор *логического отрицания* (`!`) принимает один операнд и возвращает его противоположность. Другими словами, `!true` — это `false`, а `!false` — это `true`.

Операторы логическое И (`&&`) и логическое ИЛИ (`||`) являются бинарными. Логическое И возвращает `true`, только если оба его операнда имеют значение `true`. Логическое ИЛИ возвращает `true`, если один или оба его операнда имеют значение `true`.

ПРИМЕЧАНИЕ

При чтении логическое выражение `!` произносится как «не», как в «`a` и не `b`» для выражения `a && !b`.

Поначалу логические операторы могут показаться запутанными, но очень скоро вы сможете различать их интуитивно. В листинге 2.7 приведены логические операторы.

Листинг 2.7. Программа с логическими операторами

```
#include <cstdio>

int main() {
  bool t = true;
  bool f = false;
  printf("!true: %d\n", !t); ❶
  printf("true  && false: %d\n", t && f); ❷
  printf("true  &&!false: %d\n", t &&!f); ❸
  printf("true  || false: %d\n", t || f); ❹
  printf("false || false: %d\n", f || f); ❺
}
```

```
!true: 0 ❶
true  && false: 0 ❷
true  &&!false: 1 ❸
true  || false: 1 ❹
false || false: 0 ❺
```

Здесь вы видите логическое НЕ ❶, логический оператор И ❷❸ и логический оператор ИЛИ ❹ ❺.

Тип std::byte

Системные программисты иногда работают напрямую с *необработанной памятью*, которая представляет собой набор битов без типа. В таких ситуациях следует использовать тип `std::byte`, доступный в заголовке `<cstddef>`. Тип `std::byte` раз-

решает побитовые логические операции (которые появятся в главе 7) и кое-что другое. Использование этого типа для необработанных данных, а не для целочисленного типа, может помочь избежать общих трудных для отладки ошибок программирования.

Обратите внимание, что, в отличие от большинства других фундаментальных типов в `<cstddef>`, `std::byte` не имеет точного следственного типа в языке C («тип C»). Подобно C++, в C есть типы `char` и `unsignedchar`. Эти типы менее безопасны в использовании, потому что они поддерживают много операций, которые не поддерживает `std::byte`. Например, можно выполнять арифметические действия: сложение (`+`), с `char`, но не с `std::byte`. Странно выглядящий префикс `std::` называется пространством имен, с которым мы познакомимся в «Пространствах имен» на с. 283 (сейчас просто представьте пространство имен `std::` как часть имени типа).

ПРИМЕЧАНИЕ

Существуют два подхода к тому, как произносить std. Один состоит в том, чтобы относиться к нему как к сокращению по первым буквам: «эс-ти-ди», а другой — как к акрониму: «stood» ([stu:d]). При обращении к классу в пространстве имен std ораторы обычно подразумевают оператор пространства имен `::`. Таким образом, можно было бы произнести std::byte как «stood byte» ([stu:d bʌɪt]) или, если вам не нравится краткость, как «эс-ти-ди двоеточие двоеточие байт».

Тип size_t

Тип `size_t`, также доступный в заголовке `<cstddef>`, используется для кодирования размера объектов. Объекты `size_t` гарантируют, что их максимальные значения достаточны для представления максимального размера в байтах всех объектов. Технически это означает, что `size_t` может занять 2 байта или 200 байтов в зависимости от реализации. На практике это значение обычно совпадает с `unsigned long long` на 64-битных машинах.

ПРИМЕЧАНИЕ

Тип `size_t` — это тип C в заголовке `<cstddef>`, но он идентичен версии C++, которая находится в пространстве имен `std`. Иногда вы можете увидеть (технически правильную) конструкцию `std::size_t`.

sizeof

Унарный оператор `sizeof` принимает операнд типа и возвращает размер (в байтах) этого типа. Оператор `sizeof` всегда возвращает `size_t`. Например, `sizeof(float)` возвращает количество байтов памяти, которое занимает `float`.

Спецификаторы формата

Обычными спецификаторами формата для `size_t` являются `%zu` для десятичного представления или `%zx` для шестнадцатеричного представления. Листинг 2.8 показывает, как можно проверить систему на наличие нескольких целочисленных типов.

Листинг 2.8. Программа, которая выводит размеры в байтах нескольких целочисленных типов. (Вывод поступает с 64-битной машины с Windows 10.)

```
#include <cstddef>
#include <cstdio>

int main() {
  size_t size_c = sizeof(char); ❶
  printf("char: %zu\n", size_c);
  size_t size_s = sizeof(short); ❷
  printf("short: %zu\n", size_s);
  size_t size_i = sizeof(int); ❸
  printf("int: %zu\n", size_i); size_t size_l = sizeof(long); ❹
  printf("long: %zu\n", size_l);
  size_t size_ll = sizeof(long long); ❺
  printf("long long: %zu\n", size_ll);
}
--------------------------------------------------------------------------------
char: 1 ❶
short: 2 ❷
int: 4 ❸
long: 4 ❹
long long: 8 ❺
```

Листинг 2.8 вычисляет `sizeofchar` ❶, `short` ❷, `int` ❸, `long` ❹ и `longlong` ❺ и выводит размеры, используя спецификатор формата `%zu`. Результаты будут различаться в зависимости от операционной системы. Вспомните из табл. 2.1, что каждая среда определяет свои собственные размеры для целочисленных типов. Обратите особое внимание на возвращаемое значение `long` в листинге 2.8; Linux и macOS определяют типы `long` в 8 байт.

void

Тип `void` имеет пустой набор значений. Поскольку `void`-объект не может содержать значение, C++ запрещает `void`-объекты. `void` используется в особых ситуациях, таких как тип возврата для функций, которые не возвращают никакого значения. Например, функция `taunt` не возвращает значение, поэтому тип возвращаемого значения — `void`:

```
#include <cstdio>

void taunt() {
  printf("Hey, laser lips, your mama was a snow blower.");
}
```

В главе 3 вы узнаете о других специальных способах использования `void`.

Массивы

Массивы — это последовательности переменных одинакового типа. *Типы массивов* включают в себя содержащийся тип и количество элементов в массиве. Эта информация объединена в синтаксисе объявления: тип элемента предшествует квадратным скобкам, в которых определяется размер массива.

Например, следующая строка объявляет массив из 100 объектов типа `int`:

```
int my_array[100];
```

Инициализация массива

Существует быстрый способ для инициализации массивов со значениями с использованием фигурных скобок:

```
int array[] = { 1, 2, 3, 4 };
```

Можно опустить указание длины массива, потому что она может быть выведена из числа элементов в фигурных скобках во время компиляции.

Доступ к элементам массива

Доступ к элементам массива можно получить, используя квадратные скобки, в которых указывается нужный индекс. Индексирование массива в C++ начинается с нуля, поэтому первый элемент имеет индекс 0, десятый элемент — индекс 9 и т. д. В листинге 2.9 показан пример чтения и записи элементов массива.

Листинг 2.9. Программа, индексирующая массив

```
#include <cstdio>

int main() {
  int arr[] = { 1, 2, 3, 4 }; ❶
  printf("The third element is %d.\n", arr[2] ❷);
  arr[2] = 100; ❸
  printf("The third element is %d.\n", arr[2]x); ❹
}
--------------------------------------------------------------------------------
The third element is 3. ❷
The third element is 100. ❹
```

Этот код объявляет массив из четырех элементов с именем `arr`, в котором содержатся элементы `1`, `2`, `3` и `4` ❶. На следующей строке ❷ выводится третий элемент. Затем третьему элементу назначается значение 100 ❸, поэтому, когда третий элемент выводится повторно ❹, его значение равно 100.

Экскурсия по циклу for

Цикл `for` позволяет повторять (или итерировать) выполнение оператора указанное количество раз. Можно указать отправную точку и другие условия. *Оператор инициализации* выполняется до прохождения первой итерации, поэтому в нем можно инициализировать переменные, используемые в цикле `for`. *Условное выражение* — это выражение, которое вычисляется перед каждой итерацией. Если значение равно `true`, итерация продолжается. Если `false`, цикл `for` завершается. *Оператор цикла* выполняется после каждой итерации, что полезно в ситуациях, когда необходимо увеличить переменную, чтобы охватить диапазон значений. Синтаксис цикла `for` выглядит следующим образом:

```
for(оператор-инициализации; условное-выражение; оператор-цикла) {
  --пропуск--
}
```

Например, в листинге 2.10 показано, как использовать цикл `for` для поиска элемента массива с максимальным значением.

Листинг 2.10. Поиск максимального значения, содержащегося в массиве

```
#include <cstddef>
#include <cstdio>

int main() {
  unsigned long maximum = 0; ❶
  unsigned long values[] = { 10, 50, 20, 40, 0 }; ❷
  for(size_t i=0; i < 5; i++) { ❸
    if (values[i] > maximum❹) maximum = values[i]; ❺
  }
  printf("The maximum value is %lu", maximum); ❻
}
--------------------------------------------------------------------------------
The maximum value is 50 ❻
```

Значение `maximum` ❶ инициализируется наименьшим возможным значением; здесь это 0, потому что используются беззнаковые целые числа. Затем инициализируется массив `values` ❷, проход по которому осуществляется с использованием цикла `for` ❸. Переменная цикла `i` находится в диапазоне от 0 до 4 включительно. В цикле `for` вы получаете доступ к каждому элементу значений и проверяете, больше ли значение этого элемента, чем текущее максимальное значение `maximum` ❹. Если это так, значение `maximum` переопределяется на это новое значение ❺. Когда цикл завершен, `maximum` будет равен наибольшему значению в массиве и программа выведет это значение ❻.

ПРИМЕЧАНИЕ

Если вы раньше программировали на C или C++, вам может быть интересно, почему в листинге 2.10 вместо размера `int` для типа i используется `size_t`. Учтите, что значения теоретически могут занимать максимально допустимый размер в памяти. `size_t` гарантированно сможет индексировать любое значение внутри него, а `int` — нет. На практике это не имеет большого значения, но технически выбор `size_t` является правильным.

Цикл for на основе диапазона

Из листинга 2.10 вы узнали, как использовать цикл `for` в ❸ для итерации по элементам массива. Теперь же можно избавиться от переменной цикла `i` с помощью *цикла for, основанного на диапазоне*. Для определенных объектов, таких как массивы, `for` понимает, как выполнять итерацию по диапазону значений внутри объекта. Вот синтаксис для цикла `for` на основе диапазона:

```
for(тип-элемента❶ имя-элемента❷ : имя-массива❸) {
  --пропуск--
}
```

Программа объявляет переменную цикла *имя-элемента* ❷ с типом *тип-элемента* ❶. *тип-элемента* должен соответствовать типу элементов в перебираемом массиве. Массив имеет название *имя-массива* ❸.

Листинг 2.11 — это рефакторинг листинга 2.10, где используется цикл `for` на основе диапазона.

Листинг 2.11. Рефакторинг листинга 2.10 с использованием цикла for на основе диапазона

```
#include <cstdio>

int main() {
  unsigned long maximum = 0;
  unsigned long values[] = { 10, 50, 20, 40, 0 };
  for(unsigned long value : values❶) {
    if (value❷ > maximum) maximum = value❸;
  }
  printf("The maximum value is %lu.", maximum);
}
--------------------------------------------------------------------------------
The maximum value is 50.
```

ПРИМЕЧАНИЕ

С выражениями вы познакомитесь в главе 7. А пока представьте выражение как некоторый фрагмент кода, который влияет на программу.

Листинг 2.11 значительно улучшает листинг 2.10. С первого взгляда вы можете заметить, что цикл `for` перебирает `values` ❶. Поскольку переменная цикла `i` отброшена, тело цикла `for` значительно упрощается; по этой причине можно использовать каждый элемент массива непосредственно ❷❸.

Используйте циклы `for` на основе диапазона при любой возможности.

Количество элементов в массиве

Используйте оператор `sizeof` для получения общего размера массива в байтах. Можно применить простой трюк для определения количества элементов в массиве: разделите размер массива на размер одного его элемента:

```
short array[] = { 104, 105, 32, 98, 105, 108, 108, 0 };
size_t n_elements = sizeof(array) ❶ / sizeof(short) ❷;
```

В большинстве систем `sizeof(array)` ❶ оценивается в 16 байтов, а `sizeof(short)` ❷ оценивается в 2 байта. Независимо от размера `short`, `n_elements` всегда будет инициализироваться как 8, потому что коэффициент не будет учитываться. Данное вычисление происходит во время компиляции, поэтому при расчете длины массива таким способом не существует издержек времени выполнения.

Конструкция `sizeof(x)/sizeof(y)` — это небольшой «костыль», но она широко используется в старом коде. Во второй части книги вы узнаете о других вариантах хранения данных, которые не требуют внешнего вычисления их размера. Если действительно нужно использовать массив, безопасно получить количество элементов можно, используя функцию `std::size`, доступную в заголовке `<iterator>`.

ПРИМЕЧАНИЕ

В качестве дополнительного преимущества `std::size` может использоваться с любым контейнером, предоставляющим метод `size`. Это включает в себя все контейнеры из главы 13. Такой подход особенно полезен при написании универсального кода, о котором пойдет речь в главе 6. Кроме того, программа не будет скомпилирована, если в функцию будет случайно передан неподдерживаемый тип, такой как указатель.

Строки в стиле C

Строки — это непрерывные блоки символов. Строка в стиле C или *строка с нулевым символом в конце* имеет нулевой байт, добавленный к концу строки (`null`), чтобы указать ее окончание. Поскольку элементы массива являются непрерывными, можно хранить строки в массивах символьных типов.

Строковые литералы

Строковые литералы объявляются с помощью двойных кавычек (`""`). Как и символьные литералы, строковые литералы поддерживают кодировку Unicode: просто добавьте литерал с соответствующим префиксом, например `L`. В следующем примере строковые литералы присваиваются массивам `english` и `chinese`:

```
char english[] = "A book holds a house of gold.";
char16_t chinese[] = u"\u4e66\u4e2d\u81ea\u6709\u9ec4\u91d1\u5c4b";
```

ПРИМЕЧАНИЕ

Сюрприз! Вы уже использовали строковые литералы: строки формата операторов `printf` являются строковыми литералами.

Этот код генерирует две переменные: `english`, которая содержит текст `A book holds a house of gold.`, и `chinese`, которая содержит Unicode-символы для 书中自有黄金屋.

Спецификатор формата

Спецификатор формата для узких строк (`char*`) — `%s`. Например, добавить строки в строки формата можно следующим образом:

```
#include <cstdio>

int main() {
  char house[] = "a house of gold.";
  printf("A book holds %s\n ", house);
}
--------------------------------------------------------------------------------
A book holds a house of gold.
```

ПРИМЕЧАНИЕ

Вывод Unicode-символов в консоль удивительно сложен. Как правило, нужно убедиться, что выбрана правильная кодовая страница, и эта тема выходит за рамки этой книги. Если необходимо встроить символы Unicode в строковый литерал, рассмотрите `wprintf` в заголовке `<cwchar>`.

Последовательные строковые литералы объединяются, а любые промежуточные пробелы или переводы строк игнорируются. Таким образом, можно поместить строковые литералы в несколько строк в исходном коде, и компилятор будет обрабатывать их как единое целое. Например, можно изменить этот фрагмент следующим образом:

```
#include <cstdio>

int main() {
  char house[] = "a "
      "house "
      "of " "gold.";
  printf("A book holds %s\n ", house);
}
--------------------------------------------------------------------------------
A book holds a house of gold.
```

Обычно такие конструкции полезны и удобочитаемы только тогда, когда используется длинный строковый литерал, который будет занимать несколько строк в исходном коде. Сгенерированные программы будут идентичны.

ASCII

Таблица *Американского стандартного кода обмена информацией* (American Standard Code for Information Interchange, ASCII) присваивает символам целочисленные коды. Таблица 2.4 воспроизводит таблицу ASCII. Для каждого целочисленного значения в десятичном (0d) и шестнадцатеричном (0x) виде отображается соответствующий управляющий код или печатный символ.

Таблица 2.4. Таблица ASCII

Контрольные коды			Выводимые символы								
0d	**0x**	**Код**	**0d**	**0x**	**Символ**	**0d**	**0x**	**Символ**	**0d**	**0x**	**Символ**
0	0	NULL	32	20	пробел	64	40	@	96	60	`
1	1	SOH	33	21	!	65	41	A	97	61	a
2	2	STX	34	22	"	66	42	B	98	62	b
3	3	ETX	35	23	#	67	43	C	99	63	c
4	4	EOT	36	24	$	68	44	D	100	64	d
5	5	ENQ	37	25	%	69	45	E	101	65	e
6	6	ACK	38	26	&	70	46	F	102	66	f
7	7	BELL	39	27	'	71	47	G	103	67	g
8	8	BS	40	28	(	72	48	H	104	68	h
9	9	HT	41	29	)	73	49	I	105	69	i
10	0a	LF	42	2a	*	74	4a	J	106	6a	j
11	0b	VT	43	2b	+	75	4b	K	107	6b	k
12	0c	FF	44	2c	,	76	4c	L	108	6c	l
13	0d	CR	45	2d	-	77	4d	M	109	6d	m
14	0e	SO	46	2e	.	78	4e	N	110	6e	n
15	0f	SI	47	2f	/	79	4f	O	111	6f	o
16	10	DLE	48	30	0	80	50	P	112	70	p
17	11	DC1	49	31	1	81	51	Q	113	71	q
18	12	DC2	50	32	2	82	52	R	114	72	r
19	13	DC3	51	33	3	83	53	S	115	73	s
20	14	DC4	52	34	4	84	54	T	116	74	t
21	15	NAK	53	35	5	85	55	U	117	75	u
22	16	SYN	54	36	6	86	56	V	118	76	v
23	17	ETB	55	37	7	87	57	W	119	77	w
24	18	CAN	56	38	8	88	58	X	120	78	x
25	19	EM	57	39	9	89	59	Y	121	79	y
26	1a	SUB	58	3a	:	90	5a	Z	122	7a	z
27	1b	ESC	59	3b	;	91	5b	[	123	7b	{
28	1c	FS	60	3c	<	92	5c	\	124	7c	|
29	1d	GS	61	3d	=	93	5d	]	125	7d	}
30	1e	RS	62	3e	>	94	5e	^	126	7e	~
31	1f	US	63	3f	?	95	5f	_	127	7f	DEL

Коды ASCII от 0 до 31 являются *символами кода управления*, которые управляют устройствами. В основном это анахронизмы. Когда в 1960-х годах Американская ассоциация стандартов официально оформила ASCII, современные устройства включали в себя телетайпы, устройства для чтения магнитных лент и матричные принтеры. Ниже перечислены некоторые общеупотребительные коды управления:

- 0 (NULL) используется в качестве ограничителя строки в языках программирования.
- 4 (EOT), конец передачи, завершает сеансы оболочки и связь с принтером PostScript.
- 7 (BELL) вызывает шум в устройстве.
- 8 (BS), возврат на одну позицию, заставляет устройство стирать последний символ.
- 9 (HT), горизонтальная вкладка, перемещает курсор на несколько пробелов вправо.
- 10 (LF), перевод строки, используется в качестве маркера конца строки в большинстве операционных систем.
- 13 (CR), возврат каретки, используется в сочетании с LF в качестве маркера конца строки в системах Windows.
- 26 (SUB), замещающий символ/конец файла/`ctrl-Z`, приостанавливает выполняющийся в настоящее время интерактивный процесс в большинстве операционных систем.

Остальная часть таблицы ASCII, коды от 32 до 127, — это печатные символы. Они представляют английские символы, цифры и знаки препинания.

В большинстве систем ASCII представляет тип `char`. Хотя эти отношения строго не гарантированы, фактически это считается стандартом.

Теперь пришло время объединить знания о типах `char`, массивах, циклах `for` и таблице ASCII. Листинг 2.12 показывает, как построить массив с буквами алфавита, вывести результат, а затем преобразовать этот массив в верхний регистр и снова вывести.

Сначала объявляется массив `char` длиной 27 для хранения 26 английских букв плюс символ конца строки `null` ❶. Затем перебором значений от 0 до 25 используется цикл `for` с итератором `i`. Буква *a* имеет значение 97 в таблице ASCII. Добавив 97 к итератору `i`, можно сгенерировать все строчные буквы в `alphabet` ❷. Чтобы представить `alphabet` строкой с символом `null` в конце, для `alphabet[26]` устанавливается значение `0` ❸. Затем выводится результат ❹.

Далее необходимо вывести прописные буквы алфавита. Буква *A* имеет значение `65` в таблице ASCII, поэтому каждый элемент `alphabet` стирается ❺ и снова вызывается функция `printf` ❻.

Листинг 2.12. Вывод букв алфавита в нижнем и верхнем регистрах с использованием ASCII

```
#include <cstdio>

int main() {
  char alphabet[27]; ❶
  for (int i = 0; i<26; i++) {
    alphabet[i] = i + 97; ❷
  }
  alphabet[26] = 0; ❸
  printf("%s\n", alphabet); ❹
  for (int i = 0; i<26; i++) {
    alphabet[i] = i + 65; ❺
  }
  printf("%s", alphabet); ❻
}
--------------------------------------------------------------------------------
abcdefghijklmnopqrstuvwxyz❹
ABCDEFGHIJKLMNOPQRSTUVWXYZ❻
```

Пользовательские типы

Пользовательские типы — это типы, которые пользователь может определить самостоятельно. Вот три основные категории пользовательских типов:

- **Перечисления.** Простейший из определяемых пользователем типов. Значения, которые может принимать перечисление, ограничены набором возможных значений. Перечисления отлично подходят для моделирования категориальных понятий.
- **Классы.** Более полнофункциональные типы, которые позволяют гибко объединять данные и функции. Классы, которые содержат только данные, называются простыми классами; вы узнаете о них в этом разделе.
- **Объединения.** Специализированный пользовательский тип. Все члены объединения разделяют одну и ту же ячейку памяти. Объединения опасны, и ими легко злоупотреблять.

Типы перечислений

Перечисления объявляются при помощи ключевых слов `enum class`, за которыми идут имя типа и список значений, которые он может принять. Эти значения являются произвольными буквенно-цифровыми строками, они будут определять любые категории, которые необходимо представить. В сущности, эти значения являются просто целыми числами, но они позволяют писать более безопасный и выразительный код, используя определяемые программистом типы, а не целые числа, которые могут означать что угодно. Например, в листинге 2.13 объявляется `enum class` с именем `Race`, который может принимать одно из семи значений.

Листинг 2.13. Класс перечисления, содержащий все расы из «Семиевия» Нила Стивенсона

```
enum class Race {
  Dinan,
  Teklan,
  Ivyn,
  Moiran,
  Camite,
  Julian,
  Aidan
};
```

Для инициализации переменной перечисления значением используйте имя типа, за которым следуют два двоеточия `::` и требуемое значение. Например, так можно объявить переменную `langobard_race` и инициализировать ее значением `Aidan`:

```
Race langobard_race = Race::Aidan;
```

ПРИМЕЧАНИЕ

Технически `enum class` является одним из двух видов перечислений: он называется `enum` с ограниченной областью. Для совместимости с C C++ также поддерживает `enum` с незаданной областью, который объявляется при помощи ключевого слова `enum`, а не `enum class`. Основное различие заключается в том, что перечисления с ограниченной областью требуют, чтобы тип перечисления следовал за `::` перед предшествующими значениями, тогда как перечисления с незаданной областью не требуют этого. Классы `enum` с незаданной областью менее безопасны в использовании, чем их собратья, поэтому избегайте их, если в них нет крайней необходимости. Они поддерживаются в C++ по историческим причинам, особенно во взаимодействии с кодом C. См. книгу «Эффективное использование C++» Скотта Мейерса, правило 10 для более подробной информации.

Оператор switch

Оператор ветвления (`switch`) передает управление одному из нескольких операторов в зависимости от значения *условия*, которое вычисляется как целое число или тип перечисления. Ключевое слово `switch` обозначает оператор `switch`.

Операторы `switch` обеспечивают условное ветвление. Когда выполняется оператор `switch`, управление переходит к *случаю*, соответствующему условию, или *к условию по умолчанию*, если ни один случай не соответствует условному выражению. Ключевое слово `case` обозначает случай, тогда как ключевое слово `default` обозначает условие по умолчанию.

Несколько странно то, что выполнение будет продолжаться до конца оператора `switch` или ключевого слова `break`. Практически всегда в конце каждого условия ставится `break`.

Операторы `switch` составлены из множества компонентов. Листинг 2.14 показывает, как они сочетаются друг с другом.

Листинг 2.14. Схема сочетания операторов переключателя

```
switch❶(условие❷) {
  case❸ (случай-a❹): {
    // Обработка случая a
    --пропуск--
  }❺ break❻;
  case (случай-b): {
    // Обработка случая b
    --пропуск--
  } break;
    // Обработка других условий при необходимости
    --пропуск--
  default❼: {
    // Обработка случая по умолчанию
    --пропуск--
  }
}
```

Все операторы `switch` начинаются с ключевого слова `switch` ❶, за которым следует условие в скобках ❷. Каждый случай начинается с ключевого слова `case` ❸, за которым следует перечисление или целочисленное значение ❹. Например, если *условие* ❷ равно *случай-a* ❹, то будет выполнен код в блоке, содержащем `Обработка случая a`. После каждого оператора, следующего за случаем ❺, помещается ключевое слово `break` ❻. Если условие не соответствует ни одному из случаев, обрабатывается случай по умолчанию ❼.

ПРИМЕЧАНИЕ

Скобки в объявлении каждого случая являются необязательными, но настоятельно рекомендуемыми. Без них программа иногда может вести себя удивительным образом.

Использование оператора switch с классом перечисления

В листинге 2.15 используется оператор `switch` в классе перечисления `Race` для генерации специального приветствия.

Листинг 2.15. Программа, которая печатает приветствие в зависимости от выбранного экземпляра Race

```
#include <cstdio>

enum class Race { ❶
  Dinan,
  Teklan,
  Ivyn,
  Moiran,
  Camite,
  Julian,
  Aidan
};
```

```
int main() {
  Race race = Race::Dinan; ❷

  switch(race) { ❸
  case Race::Dinan: { ❹
      printf("You work hard.");
    } break; ❺
  case Race::Teklan: {
      printf("You are very strong.");
    } break;
  case Race::Ivyn: {
      printf("You are a great leader.");
    } break;
  case Race::Moiran: {
      printf("My, how versatile you are!");
    } break;
  case Race::Camite: {
      printf("You're incredibly helpful.");
    } break;
  case Race::Julian: {
      printf("Anything you want!");
    } break;
  case Race::Aidan: {
      printf("What an enigma.");
    } break;
  default: {
      printf("Error: unknown race!"); ❻
    }
  }
}
--------------------------------------------------------------------------------
You work hard.
```

`enum class` ❶ объявляет тип перечисления `Race`, который используется для инициализации `race` значением `Dinan` ❷. Оператор `switch` ❸ оценивает условие `race`, чтобы определить, какому условию передать управление. Поскольку ранее `Race` был объявлен со значением `Dinan`, выполнение переходит к ❹, что выводит строку `You work hard`. Инструкция `break` в ❺ завершает оператор `switch`.

Условие по умолчанию `default` в ❻ было добавлено в целях безопасности. Если кто-то добавляет новое значение `Race` в классе перечисления, этот неизвестный экземпляр будет обнаружен во время выполнения и в консоли появится сообщение об ошибке.

Попробуйте поменять значение `Race` ❷ на любое другое. Как меняется результат?

Простые классы

Классы — это определяемые пользователем типы, которые содержат данные и функции, и они являются сердцем и душой C++. Самым простым видом классов являются *классы простых данных* (plain-old-data classes, POD). POD — это простые контейнеры. Их можно представить как некую неоднородную совокупность элементов потенциально *разных* типов. Каждый элемент класса называется членом.

Каждый POD начинается с ключевого слова `struct`, за которым следует желаемое имя POD. Далее вы перечисляете типы и имена членов. Рассмотрим следующее объявление класса `Book` с четырьмя членами:

```
struct Book {
  char name[256]; ❶
  int year; ❷
  int pages; ❸
  bool hardcover; ❹
};
```

Экземпляр `Book` содержит массив `char` с именем `name` ❶, `int year` ❷, `int pages` ❸ и `bool hardcover` ❹.

Переменные POD могут быть объявлены, как и любые другие переменные: по типу и имени. Затем доступ к членам переменной можно получить, используя оператор точки (.).

В листинге 2.16 используется тип `Book`.

Листинг 2.16. Пример использования POD-типа `Book` для чтения и записи членов

```
#include <cstdio>

struct Book {
  char name[256];
  int year;
  int pages;
  bool hardcover;
};

int main() {
  Book neuromancer; ❶
  neuromancer.pages = 271; ❷
  printf("Neuromancer has %d pages.", neuromancer.pages); ❸
}
--------------------------------------------------------------------------------
Neuromancer has 271 pages. ❸
```

Сначала объявляется переменная `neuromancer` ❶ типа `Book`. Затем количество страниц `neuromancer` определяется как 271 при помощи оператора точки (`.`) ❷. Наконец, код выводит сообщение и извлекает количество страниц из `neuromancer`, снова используя оператор точки ❸.

ПРИМЕЧАНИЕ

POD имеют некоторые полезные функции низкого уровня: они совместимы с C, и можно использовать машинные инструкции, которые очень эффективны для их копирования или перемещения. Также они могут быть эффективно представлены в памяти.

C++ гарантирует, что члены будут последовательно расположены в памяти, хотя некоторые реализации требуют, чтобы члены были выровнены по границам слов, что зависит от длины регистра процессора. Как правило, элементы должны быть упорядочены от самых длинных до самых коротких в определениях POD.

Объединения

Объединение является кузеном POD, который располагает всех своих членов в одном месте в памяти. Можно представить объединения как различные интерпретации блока памяти. Они могут быть полезны в некоторых ситуациях низкого уровня, например при распределении структур, которые должны быть согласованы между архитектурами, при решении проблем проверки типов, связанных с взаимодействием C/C++, и даже при упаковке битовых полей.

Листинг 2.17 показывает, как объявляется объединение: просто используйте ключевое слово `union` вместо `struct`.

Листинг 2.17. Пример объединения

```
union Variant {
  char string[10];
  int integer;
  double floating_point;
};
```

Объединение `Variant` может быть интерпретировано как `char[10]`, `int` или `double`. Оно занимает столько же памяти, сколько его самый большой член (возможно, `string` в этом случае).

Оператор точки (`.`) используется для указания интерпретации объединения. Синтаксически это похоже на доступ к члену POD, но под капотом это две совершенно разные структуры.

Поскольку все члены объединения находятся в одном месте, можно очень легко вызвать повреждение данных. Листинг 2.18 показывает эту опасность.

Листинг 2.18. Программа, использующая объединение Variant из листинга 2.17

```
#include <cstdio>

union Variant {
  char string[10];
  int integer;
  double floating_point;
};

int main() {
  Variant v; ❶
  v.integer = 42; ❷
  printf("The ultimate answer: %d\n", v.integer); ❸
  v.floating_point = 2.7182818284; ❹
  printf("Euler's number e:    %f\n", v.floating_point); ❺
  printf("A dumpster fire:     %d\n", v.integer); ❻
}
--------------------------------------------------------------------------------
The ultimate answer: 42 ❸
Euler's number e:    2.718282 ❺
A dumpster fire:     -1961734133 ❻
```

Экземпляр `Variant v` объявлен в ❶. Затем `v` интерпретируется как целое число, его значение принимается равным `42` ❷ и `v` выводится в консоль ❸. Затем `v` повторно интерпретируется как число с плавающей точкой и его значение переназначается ❹. Программа выводит это в консоль, и все выглядит хорошо ❺. Все идет нормально.

Катастрофа случается только тогда, когда вы пытаетесь снова интерпретировать `v` как целое число ❻. Исходное значение `v (42)` ❷ было повреждено при назначении номера Эйлера ❹.

Это главная проблема объединений: нужно следить за тем, какая интерпретация уместна. Компилятор в этом случае не поможет.

Следует избегать использования объединений во всех случаях, кроме самых редких, и в этой книге они больше не встретятся. В разделе «variant» на с. 454 обсуждаются некоторые более безопасные варианты, когда требуется использовать функциональность нескольких типов.

Полнофункциональные классы в C++

Классы POD содержат только элементы данных, и иногда это все, что нужно от самого класса. Однако разработка программы с использованием только POD может создать много сложностей. С одной из таких сложностей можно бороться при помощи *инкапсуляции*, схемы проектирования, которая связывает данные с функциями, которые ими управляют. Объединение связанных функций и данных помогает упростить код по крайней мере двумя способами. Во-первых, можно поместить связанный код в одном месте, что поможет изучать работу программы. Вы можете понять, как работает сегмент кода, потому что он описывает как состояние программы, так и то, как код изменяет это состояние. Во-вторых, можно скрыть часть кода и данных класса от остальной части программы, используя практику, называемую *сокрытием информации*.

В C++ инкапсуляция достигается путем добавления методов и элементов управления доступом к определениям классов.

Методы

Методы являются функциями-членами. Они создают явную связь между классом, его членами данных и некоторым кодом. Определение метода так же просто, как добавление функции в определение класса. Методы имеют доступ ко всем членам класса.

Рассмотрим пример класса `ClockOfTheLongNow`, который отслеживает значение года. Здесь определены член `int year` и метод `add_year`, который его увеличивает:

```
struct ClockOfTheLongNow {
  void add_year() { ❶
    year++; ❷
  }
  int year; ❸
};
```

Объявление метода `add_year` ❶ выглядит как любая другая функция, которая не принимает параметров и не возвращает значений. В рамках метода переменная-член `year` ❸ увеличивается на 1 ❷. Листинг 2.19 показывает, как можно использовать класс для отслеживания года.

Листинг 2.19. Программа, использующая struct ClockOfTheLongNow

```
#include <cstdio>

struct ClockOfTheLongNow {
  --пропуск--
};


int main() {
  ClockOfTheLongNow clock; ❶
  clock.year = 2017; ❷
  clock.add_year(); ❸
  printf("year: %d\n", clock.year); ❹
  clock.add_year(); ❺
  printf("year: %d\n", clock.year); ❻
}
--------------------------------------------------------------------------------
year: 2018 ❹
year: 2019 ❻
```

Вы объявляете экземпляр `ClockOfTheLongNowclock` ❶, а затем устанавливаете `year` в `clock` равным `2017` ❷. Затем вызывается метод `add_year` для `clock` ❸ и выводится значение `clock.year` ❹. Программа завершается, увеличивая значение на 1 ❺ и еще раз выводя его ❻.

Контроль доступа

Контроль доступа ограничивает доступ к членам класса. Основные средства контроля доступа — это *публичность* и *приватность* членов. Кто угодно может получить доступ к публичному члену, но только сам класс может получить доступ к своим приватным членам. Все члены `struct` по умолчанию публичны.

Приватные члены играют важную роль в инкапсуляции. Рассмотрим снова класс `ClockOfTheLongNow`. Доступ к члену `year` можно получить из любого места — как для чтения, так и для записи. Предположим, необходимо защитить `year` от принятия значения меньше 2019. Можно сделать это в два этапа: сделать год приватным и потребовать от любого, кто использует класс (потребителей), взаимодействовать с `year` только через методы класса. Листинг 2.20 показывает этот подход.

В `ClockOfTheLongNow` были добавлены два метода: *сеттер* ❶ и *геттер* ❸ для `year`. Вместо того чтобы позволить пользователю `ClockOfTheLongNow` изменять `year` напрямую, вы устанавливаете `year` с помощью `set_year`. Это добавление проверки входных данных гарантирует, что `new_year` никогда не будет меньше `2019` ❷. Если это так, код возвращает `false` и оставляет `year` неизмененным. В противном случае

year обновляется и возвращает true. Чтобы получить значение year, пользователь вызывает get_year.

Листинг 2.20. Обновленный ClockOfTheLongNow из листинга 2.19, который инкапсулирует year

```
struct ClockOfTheLongNow {
  void add_year() {
    year++;
  }
  bool set_year(int new_year) { ❶
    if (new_year < 2019) return false; ❷
    year = new_year;
    return true;
  }
  int get_year() { ❸
    return year;
}
private: ❹
  int year;
};
```

Вы использовали метку контроля доступа private ❹, чтобы запретить пользователям доступ к year. Теперь пользователи могут получить доступ к year только из ClockOfTheLongNow.

Ключевое слово class

Можно заменить ключевое слово struct на class, которое по умолчанию объявляет члены закрытыми. Помимо контроля доступа по умолчанию, классы, объявленные с ключевыми словами struct и class, одинаковы. Например, можно объявить ClockOfTheLongNow следующим образом:

```
class ClockOfTheLongNow {
  int year;
public:
  void add_year() {
    --пропуск--
  }
  bool set_year(int new_year) {
    --пропуск--
  }
  int get_year() {
    --пропуск--
  }
};
```

Способ объявления классов зависит от стиля. Нет абсолютно никакой разницы между struct и class, кроме контроля доступа по умолчанию. Я предпочитаю использовать ключевое слово struct, потому что мне нравится, когда публичные члены перечислены первыми. Но в реальности встречаются все виды соглашений. Развивайте собственный стиль и придерживайтесь его.

Инициализация членов

После инкапсуляции `year` необходимо использовать методы для взаимодействия с `ClockOfTheLongNow`. В листинге 2.21 показано, как можно объединить эти методы в программу, которая пытается поменять значение `year` на `2018`. Это не удается, и затем программа определяет значение `2019`, увеличивает `year` и выводит его окончательное значение.

Листинг 2.21. Программа, которая использует ClockOfTheLongNow

```
#include <cstdio>

struct ClockOfTheLongNow {
  --пропуск--
};

int main() {
  ClockOfTheLongNow clock; ❶
  if(!clock.set_year(2018)) { ❷ // не удастся; 2018 < 2019
    clock.set_year(2019); ❸
  }
  clock.add_year(); ❹
  printf("year: %d", clock.get_year());
}
--------------------------------------------------------------------------------
year: 2020 ❺
```

Код объявляет экземпляр `clock` ❶ и пытается установить его `year` как `2018` ❷. Этот вариант проваливается, потому что `2018` меньше `2019`, и программа затем меняет значение `year` на `2019` ❸. `year` увеличивается один раз ❹, а затем программа выводит его значение.

В главе 1 вы увидели, как неинициализированные переменные могут содержать случайные данные при прохождении через отладчик. Структура `ClockOfTheLongNow` имеет ту же проблему: когда переменная `clock` объявлена ❶, `year` еще не инициализирована. Нужно гарантировать, что `year` никогда не будет меньше 2019, *ни при каких обстоятельствах*. Такое требование называется *инвариантом класса* — свойство класса, которое всегда истинно (то есть никогда не изменяется).

В этой программе `clock` в конечном итоге переходит в допустимое состояние ❸, но можно добиться большего, используя *конструктор*. Конструкторы инициализируют объекты и применяют инварианты классов с самого начала жизни объекта.

Конструкторы

Конструкторы — это специальные методы со специальными объявлениями. Объявления конструктора не указывают тип возвращаемого значения, а их имя соответствует имени класса. Например, конструктор в листинге 2.22 не принимает

аргументов и устанавливает значение `year` в `2019`, что приводит к тому, что `year` по умолчанию равен `2019`.

Листинг 2.22. Улучшение листинга 2.21 с помощью конструктора без параметров

```
#include <cstdio>

struct ClockOfTheLongNow {
  ClockOfTheLongNow() { ❶
    year = 2019; ❷
  }
  --пропуск--
};

int main() {
  ClockOfTheLongNow clock; ❸
  printf("Default year: %d", clock.get_year()); ❹
}
--------------------------------------------------------------------------------
Default year: 2019 ❹
```

Конструктор не принимает аргументов ❶ и устанавливает `year` в `2019` ❷. При объявлении нового `ClockOfTheLongNow` ❸ `year` по умолчанию равен `2019`. Программа получает доступ к `year` с помощью `get_year` и выводит значение в консоль ❹.

А если нужно инициализировать `ClockOfTheLongNow` с пользовательским значением `year`? Конструкторы могут принимать любое количество аргументов. Можно реализовать сколько угодно конструкторов, если типы их аргументов различаются.

Рассмотрите пример в листинге 2.23, где добавляется конструктор, принимающий `int`. Конструктор инициализирует `year` значением аргумента.

Листинг 2.23. Новый вариант листинга 2.22 с другим конструктором

```
#include <cstdio>

struct ClockOfTheLongNow {
  ClockOfTheLongNow(int year_in) { ❶
    if(!set_year(year_in)) { ❷
      year = 2019; ❸
    }
  }
  --пропуск--
};

int main() {
  ClockOfTheLongNow clock{ 2020 }; ❹
  printf("Year: %d", clock.get_year()); ❺
}
--------------------------------------------------------------------------------
Year: 2020 ❺
```

Новый конструктор ❶ принимает один аргумент `year_in` типа `int`. Вы вызываете `set_year` с помощью `year_in` ❷. Если `set_year` возвращает `false`, вызывающий код предоставил неверный ввод и `year_in` переопределяется значением по умолчанию 2019 ❸. В методе `main` создается экземпляр `clock` с новым конструктором ❹, а затем выводится результат ❺.

Магия `ClockOfTheLongNow clock {2020};` называется инициализацией.

ПРИМЕЧАНИЕ

Вам может не понравиться идея, что недействительные экземпляры `year_in` были незаметно исправлены на `2019` ❸. Мне это тоже не нравится. Исключения решают эту проблему; о них вы узнаете в подразделе «Исключения» на с. 158.

Инициализация

Инициализация объектов, или просто *инициализация*, — это то, как объекты воплощаются в жизнь. К сожалению, синтаксис инициализации объекта сложен. К счастью, процесс инициализации прост. Этот раздел превращает бурлящий котел инициализации объекта в C++ в приятный рассказ.

Инициализация базового типа нулевым значением

Начнем с инициализации объекта базового типа нулем. Есть четыре способа сделать это:

```
int a = 0;   ❶ // Инициализируется значением 0
int b{};     ❷ // Инициализируется значением 0
int c = {};  ❸ // Инициализируется значением 0
int d;       ❹ // Инициализируется значением 0 (возможно)
```

Три из них надежны: явно установите значение с помощью литерала ❶, используйте фигурные скобки `{}` ❷ или метод равно + фигурные скобки `= {}` ❸. Объявление объекта без дополнительной записи ❹ ненадежно; это работает только в определенных ситуациях. Даже если вы знаете, каковы эти ситуации, следует избегать этого поведения, потому что оно вносит путаницу.

Использование фигурных скобок `{}` для инициализации переменной ожидаемо называется *фигурной инициализацией*. Частично причина разного синтаксиса инициализации C++ заключается в том, что язык вырос из C, где жизненные циклы объектов примитивны, в язык с устойчивым и функциональным жизненным циклом объектов. Разработчики языка включили инициализацию в современный C++, чтобы сгладить острые углы синтаксиса инициализации. Короче говоря, независимо от области видимости или типа объекта *инициализация фигурными скобками всегда применима*, тогда как другие обозначения — нет. Позже в этой главе вы узнаете общее правило, которое поощряет повсеместное использование скобок для инициализации.

Инициализация базового типа произвольным значением

Инициализация в произвольное значение аналогична инициализации базового типа нулем:

```
int e = 42;      ❶ // Инициализируется значением 42
int f{ 42 };     ❷ // Инициализируется значением 42
int g = { 42 }; ❸ // Инициализируется значением 42
int h(42);       ❹ // Инициализируется значением 42
```

Есть четыре способа инициализации: со знаком равенства ❶, с фигурными скобками ❷, со знаком равенства и фигурными скобками ❸ и со скобками ❹. Все они производят идентичный код.

Инициализация классов POD

Обозначение для инициализации POD в основном следует примеру с базовыми типами. Листинг 2.24 показывает сходство, объявляя тип POD, содержащий три члена, и инициализируя его экземпляры различными значениями.

Листинг 2.24. Программа, где показаны различные способы инициализации POD

```
#include <cstdint>

struct PodStruct {
  uint64_t a;
  char b[256];
  bool c;
};

int main() {
  PodStruct initialized_pod1{};    ❶  // Все поля приравниваются к нулю
  PodStruct initialized_pod2 = {}; ❷  // Все поля приравниваются к нулю

  PodStruct initialized_pod3{ 42, "Hello" };  ❸      // Поля a и b заданы, c = 0
  PodStruct initialized_pod4{ 42, "Hello", true }; ❹ // Все поля заданы
}
```

Инициализация объекта POD нулем аналогична инициализации объектов базовых типов нулем. Подходы со скобками ❶ и знаком равенства со скобками ❷ дают один и тот же код: поля инициализируются нулями.

ПРЕДУПРЕЖДЕНИЕ

Нельзя использовать метод приравнивания к нулю в POD. Следующий код не будет компилироваться, потому что это прямо запрещено в правилах языка:

```
PodStruct initialized_pod = 0;
```

Инициализация POD произвольными значениями

Можно инициализировать поля произвольными значениями, используя инициализаторы в фигурных скобках. Аргументы внутри инициализаторов в скобках

должны соответствовать типам членов POD. Порядок аргументов слева направо соответствует порядку членов сверху вниз. Любые пропущенные члены обнуляются. Члены `a` и `b` инициализируются значениями 42 и `Hello` после инициализации `initialized_pod3` ❸, а `c` обнуляется (устанавливается в `false`), потому что она была пропущена в фигурной инициализации. Инициализация `initialized_pod4` ❹ включает аргумент для `c` (`true`), поэтому значение этого члена устанавливается равным `true` после инициализации.

Инициализация знаком равенства и скобками работает идентично. Например, можно заменить ❹ следующим образом:

```
PodStruct initialized_pod4 = { 42, "Hello", true };
```

Можно пропустить поля только справа налево, поэтому следующий фрагмент не будет компилироваться:

```
PodStruct initialized_pod4 = { 42, true };
```

ПРЕДУПРЕЖДЕНИЕ

Нельзя использовать скобки для инициализации POD. Следующий код не скомпилируется:

```
PodStruct initialized_pod(42, "Hello", true);
```

Инициализация массивов

Массивы инициализируются как классы POD. Основное различие между объявлениями массива и POD — это то, что для массивов указывается размер. Напомним, что этот аргумент заключен в квадратные скобки `[]`.

При использовании фигурных инициализаторов для инициализации массивов аргумент размера становится необязательным; компилятор может вывести его из числа аргументов инициализатора.

Листинг 2.25 показывает некоторые способы инициализации массива.

Листинг 2.25. Программа, в которой перечислены различные способы инициализации массива

```
int main() {
  int array_1[]{ 1, 2, 3 };  ❶ // Массив размером 3; 1, 2, 3
  int array_2[5]{};          ❷ // Массив размером 5; 0, 0, 0, 0, 0
  int array_3[5]{ 1, 2, 3 }; ❸ // Массив размером 5; 1, 2, 3, 0, 0
  int array_4[5];            ❹ // Массив размером 5; неинициализированные значения
}
```

Массив `array_1` имеет размер три, а его элементы равны 1, 2 и 3 ❶. Массив `array_2` имеет размер пять, поскольку был указан аргумент размера ❷. Инициализатор в фигурных скобках пуст, поэтому все пять элементов инициализируются нулями.

Массив `array_3` также имеет размер пять, но инициализатор со скобками не пустой. Он содержит три элемента, поэтому оставшиеся два элемента инициализируются нулями ❸. Массив `array_4` не имеет инициализатора в скобках, поэтому он содержит неинициализированные объекты ❹.

ПРЕДУПРЕЖДЕНИЕ

Инициализируется ли `array_4` или нет, зависит от тех же правил, что и при инициализации базового типа. Длительность хранения объекта, о которой вы узнаете в разделе «Длительность хранения объекта» на с. 148, определяет правила. Не нужно запоминать эти правила, если вы явно говорите об инициализации.

Полнофункциональные классы

В отличие от базовых типов и POD, полнофункциональные классы *всегда инициализируются*. Другими словами, один из полнофункциональных конструкторов класса всегда вызывается во время инициализации. Какой конструктор вызывается, зависит от аргументов, заданных во время инициализации.

Класс в листинге 2.26 помогает разъяснить, как использовать полнофункциональные классы.

Листинг 2.26. Класс, объявляющий, какой из его нескольких конструкторов вызывается во время инициализации

```
#include <cstdio>

struct Taxonomist {
  Taxonomist() { ❶
    printf("(no argument)\n");
  }
  Taxonomist(char x) { ❷
    printf("char: %c\n", x);
  }
  Taxonomist(int x) { ❸
    printf("int: %d\n", x);
  }
  Taxonomist(float x) { ❹
    printf("float: %f\n", x);
  }
};
```

Класс `Taxonomist` u имеет четыре конструктора. Если не указывать параметры, будет вызван конструктор без параметров ❶. Если предоставить `char`, `int` или `float`, то во время инициализации, будет вызван соответствующий конструктор: ❷, ❸ или ❹ соответственно. В каждом случае конструктор предупреждает вас с помощью оператора `printf`.

Листинг 2.27 инициализирует несколько экземпляров класса `Taxonomist` разных синтаксисов и параметров.

Листинг 2.27. Программа, использующая класс Taxonomist с различными синтаксисами инициализации

```
#include <cstdio>

struct Taxonomist {
  --пропуск--
};

int main() {
  Taxonomist t1; ❶
  Taxonomist t2{ 'c' }; ❷
  Taxonomist t3{ 65537 }; ❸
  Taxonomist t4{ 6.02e23f }; ❹
  Taxonomist t5('g'); ❺
  Taxonomist t6 = { 'l' }; ❻
  Taxonomist t7{}; ❼
  Taxonomist t8(); ❽
}
--------------------------------------------------------------------------------
(no argument) ❶
char: c ❷
int: 65537 ❸
float: 602000017271895229464576.000000 ❹
char: g ❺
char: l ❻
(no argument) ❼
```

Без круглых и фигурных скобок вызывается конструктор без параметров ❶. В отличие от POD и базовых типов, вы можете положиться на эту инициализацию независимо от того, где был объявлен объект. С инициализаторами в скобках конструкторы `char` ❷, `int` ❸ и `float` ❹ вызываются, как и ожидалось. Также можно использовать круглые скобки ❺ и равно с фигурными скобками ❻; они вызывают ожидаемые конструкторы.

Хотя полнофункциональные классы всегда инициализируются, некоторым программистам нравится единообразие использования одного и того же синтаксиса инициализации для всех объектов. Это не проблема при использовании инициализации со скобками; конструктор по умолчанию вызывается, как ожидалось ❼.

К сожалению, использование круглых скобок ❽ вызывает некоторое удивительное поведение. Программа ничего не возвращает.

Если бегло посмотреть, эта инициализация ❽ выглядит как объявление функции, и это так и есть. Из-за некоторых тайных правил синтаксического анализа языка вы объявили компилятору, что функция `t8`, которую еще предстоит определить, не принимает аргументов и возвращает объект типа `Taxonomist`. Упс!

ПРИМЕЧАНИЕ

Раздел «Объявления функций» на с. 311 более подробно описывает эту операцию. Но сейчас просто знайте, что можно предоставить объявление функции, которое определяет модификаторы функции, имя, параметры и тип возвращаемого значения, а затем позже предоставить тело в его определении.

Эта широко известная проблема называется *самым неприятным анализом*, и это главная причина, по которой сообщество C++ добавило в язык синтаксис фигурной инициализации. *Сужение преобразований* — вот еще одна проблема.

Сужение преобразований

Инициализация со скобками будет генерировать предупреждения всякий раз, когда встречаются неявные сужающие преобразования. Это хорошая функция, которая может уберечь вас от ошибок. Рассмотрим следующий пример:

```
float a{ 1 };
float b{ 2 };
int narrowed_result(a/b); ❶  // Потенциально опасное сужение преобразований
int result{ a/b };        ❷  // Компилятор выдает предупреждение
```

Разделение двух литералов типа `float` приводит к получению числа с плавающей точкой. При инициализации `narrowed_result` ❶ компилятор молча сужает результат `a/b (0, 5)` до `0`, потому что были использованы скобки `()` для инициализации. При использовании фигурных инициализаторов компилятор выдает предупреждение ❷.

Инициализация членов класса

Можно использовать инициализацию со скобками для инициализации членов класса, как показано здесь:

```
struct JohanVanDerSmut {
  bool gold = true; ❶
  int year_of_smelting_accident{ 1970 }; ❷
  char key_location[8] = { "x-rated" }; ❸
};
```

Член `gold` инициализируется с использованием инициализации со знаком равенства ❶, `year_of_smelting_accident` — с помощью фигурной инициализации ❷, а `key_location` — с использованием фигурных скобок и равенства ❸. Нельзя использовать скобки для инициализации переменных-членов.

Приготовьтесь

Варианты инициализации объектов сбивают с толку даже опытных программистов на C++. Вот простое правило инициализации: *везде используйте фигурные скобки*. Инициализаторы с фигурными скобками работают, как и предполагалось, почти везде, и они вызывают меньше всего сюрпризов. По этой причине фигурная инициализация также называется *равномерной инициализацией*. Остальная часть книги следует этому руководству.

ПРЕДУПРЕЖДЕНИЕ

В определенных классах в C++ stdlib вы нарушите правило использования инициализированных фигурными скобками. Часть 2 подробно пояснит эти исключения из правила.

Деструктор

Деструктор объекта — это его функция очистки. Деструктор вызывается до уничтожения объекта. Деструкторы почти никогда не вызываются явно: компилятор будет гарантировать, что деструктор каждого объекта будет вызван соответствующим образом. Деструктор класса объявляется с помощью тильды `~`, за которой следует имя класса.

В следующем классе `Earth` есть деструктор, который выводит `Making way for hyperspace bypass`:

```
#include <cstdio>

struct Earth {
  ~Earth() { // Деструктор класса Earth
       printf("Making way for hyperspace bypass");
  }
}
```

Определение деструктора необязательно. Если вы решите реализовать деструктор, он не должен принимать никаких параметров. Примеры действий, которые можно предпринять в деструкторе, включают освобождение файловых дескрипторов, очистку сетевых сокетов и освобождение динамических объектов.

Если деструктор не определен явным образом, автоматически создается деструктор по умолчанию. Поведение деструктора по умолчанию — не выполнять никаких действий.

Вы узнаете намного больше о деструкторах в разделе «Отслеживание жизненного цикла объекта» на с. 156.

Итоги

Эта глава представила основы C++ — его систему типов. Вы узнали о базовых типах, строительных блоках всех других типов. Затем продолжили работу с пользовательскими типами, включая класс `enum`, классы POD и полнофункциональные классы C++. Экскурсия по классам завершилась обсуждением конструкторов, синтаксиса инициализации и деструкторов.

Упражнения

2.1. Создайте `enum class Operation` («Операция»), который имеет значения `Add` («Добавить»), `Subtract` («Отнять»), `Multiply` («Умножить») и `Divide` («Разделить»).

2.2. Создайте `struct Calculator` («Калькулятор»). Он должен иметь единственный конструктор, который принимает `Operation`.

2.3. Создайте в `Calculator` метод с именем `int calc (int a, int b)`. После вызова этот метод должен выполнить сложение, вычитание, умножение или деление на основе параметра своего конструктора и вернуть результат.

2.4. Поэкспериментируйте с различными средствами инициализации экземпляров класса `Calculator`.

Что еще почитать?

- Международный стандарт ИСО/МЭК (2017) — Язык программирования C++ (Международная организация по стандартизации; Женева, Швейцария; isocpp.org/std/the-standard/)
- «Язык программирования C++», 4-е издание, Бьёрн Страуструп (Бином, 2011)
- «Эффективное использование C++», Скотт Мейерс (ДМК, 2017)
- «Эффективное программирование на C++», Эндрю Кениг, Барбара Му (Вильямс, 2002)

3 Ссылочные типы

Отладка кода вдвое сложнее, чем его написание. Так что если вы пишете код настолько умно, насколько можете, то вы по определению недостаточно сообразительны, чтобы его отлаживать.

Брайан Керниган

Ссылочные типы хранят адреса памяти объектов. Эти типы обеспечивают эффективное программирование, и многие из них имеют элегантный дизайн. В этой главе я рассмотрю два вида ссылочных типов: указатели и ссылки. По ходу дела обсудим `this`, `const` и `auto`.

Указатели

Указатели — это основной механизм, используемый для обращения к адресам памяти. Указатели кодируют обе части информации, необходимые для взаимодействия с другим объектом, то есть адрес объекта и тип объекта.

Можно объявить тип указателя, добавив звездочку (`*`) к указанному типу. Например, указатель на `int` с именем `my_ptr` объявляется следующим образом:

```
int* my_ptr;
```

Спецификатор формата для указателя — `%p`. Например, чтобы вывести значение в `my_ptr`, можно использовать следующее:

```
printf("The value of my_ptr is %p.", my_ptr);
```

Указатели являются объектами очень низкого уровня. Хотя они играют центральную роль в большинстве программ на C, C++ предлагает высокоуровневые, иногда более

эффективные конструкции, которые устраняют необходимость непосредственно работать с адресами памяти. Тем не менее указатели являются фундаментальной концепцией, с которой вы, несомненно, столкнетесь в своих путешествиях по системному программированию.

В этом разделе вы узнаете, как найти адрес объекта и как присвоить результат переменной-указателю. Вы также узнаете, как выполнить противоположную операцию, которая называется *разыменованием*: по указателю можно получить объект, находящийся по соответствующему адресу.

Вы узнаете больше о *массивах*, простейшей конструкции для управления коллекцией объектов, а также о том, как массивы связаны с указателями. Как низкоуровневые конструкции массивы и указатели относительно опасны. Вы узнаете о том, что может пойти не так, когда программы на основе указателей и массивов не работают.

В этой главе представлены два специальных вида указателей: указатели `void` и указатели `std::byte`. Эти очень полезные типы имеют особое поведение, о котором нужно помнить. Кроме того, вы узнаете, как кодировать пустые указатели с помощью `nullptr` и как использовать указатели в логических выражениях, чтобы определить, являются ли они пустыми.

Обращение к переменным

Адрес переменной можно получить, предварительно добавив *оператор вычисления адреса* (`&`). Возможно, понадобится использовать этот оператор для инициализации указателя, чтобы он «указывал» на соответствующую переменную. Такие требования к программированию возникают очень часто при разработке операционных систем. Например, основные операционные системы, такие как Windows, Linux и FreeBSD, имеют интерфейсы, которые интенсивно используют указатели.

Листинг 3.1 демонстрирует, как получить адрес `int`.

Листинг 3.1. Программа, которая использует оператор вычисления адреса

```
#include <cstdio>

int main() {
  int gettysburg{}; ❶
  printf("gettysburg: %d\n", gettysburg); ❷
  int *gettysburg_address = &gettysburg; ❸
  printf("&gettysburg: %p\n", gettysburg_address); ❹
}
```

Сначала объявляется целое число `gettysburg` ❶ и выводится его значение ❷. Затем объявляется указатель, называемый `gettysburg_address`, на адрес этого целого числа ❸; обратите внимание, что звездочка стоит перед указателем, а амперсанд — перед `gettysburg`. Наконец, указатель выводится на экран ❹, чтобы показать адрес целого числа `gettysburg`.

Если вы запустите листинг 3.1 в Windows 10 x86, то увидите следующий вывод:

```
gettysburg: 0
&gettysburg: 0053FBA8
```

Запуск того же кода в Windows 10 x64 приводит к следующему выводу:

```
gettysburg: 0
&gettysburg: 0000007DAB53F594
```

Ваш вывод должен иметь идентичное значение для `gettysburg`, но `gettysburg_address` должен каждый раз отличаться. Это изменение связано со *случайным распределением адресного пространства*, которое представляет собой функцию безопасности, что запутывает базовый адрес важных областей памяти, тем самым затрудняя эксплуатацию.

> **СЛУЧАЙНОЕ РАСПРЕДЕЛЕНИЕ АДРЕСНОГО ПРОСТРАНСТВА**
>
> Почему случайное распределение адресного пространства препятствует эксплуатации? Когда хакер находит в программе эксплуатируемое условие, он может иногда втиснуть вредоносную нагрузку в предоставленный пользователем ввод. Одна из первых функций безопасности, предназначенная для предотвращения запуска этой вредоносной полезной нагрузки, заключается в том, чтобы сделать все разделы данных неисполняемыми. Если компьютер пытается выполнить данные в виде кода, то теория состоит в том, что он знает, что что-то не так, и должен завершить программу с исключением.
>
> Некоторые чрезвычайно умные хакеры придумали, как совершенно непредвиденным образом перепрофилировать инструкции исполняемого кода, тщательно продумав эксплойты, содержащие так называемые программы, ориентированные на возврат. Эти эксплойты могут быть организованы для вызова соответствующих системных API, чтобы пометить их исполняемый файл полезной нагрузки, что препятствует уменьшению неисполняемой памяти.
>
> Случайное распределение адресного пространства борется с программированием, ориентированным на возврат, путем случайного распределения адресов памяти, что затрудняет перепрофилирование существующего кода, поскольку злоумышленник не знает, где он находится в памяти.

Также обратите внимание, что в выходных данных для листинга 3.1 `gettysburg_address` содержит 8 шестнадцатеричных цифр (4 байта) для архитектуры x86 и 16 шестнадцатеричных цифр (8 байтов) для архитектуры x64. Это должно иметь некоторый смысл, потому что в современных ПК размер указателя такой же, как универсальный регистр центрального процессора. Архитектура x86 имеет 32-битные (4-байтовые) универсальные регистры, тогда как архитектура x64 — 64-битные (8-байтовые).

Разыменование указателей

Оператор разыменования (*) — это унарный оператор, обращающийся к объекту, на который ссылается указатель. Это операция обратна операции вычисления адреса.

По указанному адресу можно получить находящийся там объект. Системные программисты очень часто используют оператор разыменования, как и оператор вычисления адреса. Многие API операционной системы будут возвращать указатели, и если нужно получить доступ к объекту, на который указывает ссылка, будет использоваться оператор разыменования.

К сожалению, оператор разыменования может вызвать много путаницы с нотацией у новичков, потому что и оператор разыменования, и объявление указателя, и операция умножения используют звездочки. Помните, что звездочка добавляется к концу типа объекта, на который нужно направить указатель; однако оператор разыменования — звездочку — нужно добавлять к указателю, например так:

```
*gettysburg_address
```

Получив доступ к объекту, добавив оператор разыменования к указателю, можно обработать результат как любой другой объект указанного типа. Например, поскольку `gettysburg` является целым числом, можно записать значение 17325 в `gettysburg` с помощью `gettysburg_address`. Правильный синтаксис выглядит следующим образом:

```
*gettysburg_address = 17325;
```

Поскольку разыменованный указатель, то есть `*gettysburg_address`, появляется слева от знака равенства, происходит запись по адресу, где хранится `gettysburg`.

Если разыменованный указатель появляется где-либо, кроме левой части равенства, то выполняется чтение по адресу. Чтобы получить `int`, на который указывает `gettysburg_address`, достаточно просто воспользоваться оператором разыменования. Например, следующий оператор выведет значение, сохраненное в `gettysburg`:

```
printf("%d", *gettysburg_address);
```

В листинге 3.2 используется оператор разыменования для чтения и записи.

Листинг 3.2. Пример программы, где показаны чтение и запись с использованием указателя (выходные данные получены с компьютера с Windows 10 x64)

```
#include <cstdio>

int main() {
  int gettysburg{};
  int* gettysburg_address = &gettysburg; ❶
  printf("Value at gettysburg_address: %d\n", *gettysburg_address); ❷
  printf("Gettysburg Address: %p\n", gettysburg_address); ❸
  *gettysburg_address = 17325; ❹
  printf("Value at gettysburg_address: %d\n", *gettysburg_address); ❺
  printf("Gettysburg Address: %p\n", gettysburg_address); ❻
}
--------------------------------------------------------------------------------
Value at gettysburg_address: 0 ❷
Gettysburg Address: 000000B9EEEFFB04 ❸
Value at gettysburg_address: 17325 ❺
Gettysburg Address: 000000B9EEEFFB04 ❻
```

Сначала `gettysburg` инициализируется нулем. Затем указатель `gettysburg_address` инициализируется адресом `gettysburg` ❶. Затем на экран выводятся `int`, на который указывает `gettysburg_address` ❷, и само значение `gettysburg_address` ❸.

Значение 17325 записывается в память, на которую указывает `gettysburg_address` ❹, а затем заново выводятся указанное значение ❺ и адрес ❻.

Листинг 3.2 будет функционально идентичным, если вы присвоите значение 17325 непосредственно `gettysburg` вместо указателя `gettysburg_address`, например:

```
gettysburg = 17325;
```

Этот пример показывает тесную связь между указанным объектом (`gettysburg`) и разыменованным указателем на этот объект (`*gettysburg_address`).

Оператор «стрелка»

Оператор «стрелка» (`->`) выполняет две одновременные операции:

- разыменовывает указатель;
- получает доступ к члену указанного объекта.

Этот оператор можно использовать, чтобы уменьшить *условные помехи*, то есть сопротивление, которое программист испытывает, выражая свое намерение в коде, при обработке указателей на классы. Вам нужно будет обрабатывать указатели на классы в различных шаблонах проектирования. Например, можно передать указатель на класс в качестве параметра функции. Если принимающая функция должна взаимодействовать с членом этого класса, оператор «стрелка» — подходящий инструмент для работы.

В листинге 3.3 используется -> для чтения значения переменной `year` из объекта `ClockOfTheLongNow` (который был реализован в листинге 2.22).

Листинг 3.3. Использование указателя и оператора «стрелка» для управления объектом ClockOfTheLongNow (выходные данные получены с компьютера с Windows 10 x64)

```
#include <cstdio>

struct ClockOfTheLongNow {
  --пропуск--
};

int main() {
  ClockOfTheLongNow clock;
  ClockOfTheLongNow* clock_ptr = &clock; ❶
  clock_ptr->set_year(2020); ❷
  printf("Address of clock: %p\n", clock_ptr); ❸
  printf("Value of clock's year: %d", clock_ptr->get_year()); ❹
}
--------------------------------------------------------------------------------
Address of clock: 000000C6D3D5FBE4 ❸
Value of clock's year: 2020 ❹
```

Сначала объявляется переменная `clock_ptr` ❶, а затем ее адрес сохраняется в `clock_ptr`. Затем используется оператор «стрелка», чтобы установить значение члена `year` в `clock` равным 2020 ❷. Наконец, выводятся адрес `clock` ❸ и значение `year` ❹.

Можно достичь идентичного результата, используя операторы разыменования (`*`) и члена (`.`). Например, можно было бы написать последнюю строку листинга 3.3 следующим образом:

```
printf("Value of clock's year: %d", (*clock_ptr).get_year());
```

Сначала происходит разыменование `clock_ptr`, а затем код получает доступ к `year`. Хотя это эквивалентно вызову оператора «стрелка», это более подробный синтаксис, и он не дает преимущества по сравнению с его более простой альтернативой.

ПРИМЕЧАНИЕ

На данный момент используйте скобки, чтобы подчеркнуть порядок операций. В главе 7 рассматриваются правила приоритета операторов.

Указатели и массивы

Указатели имеют несколько общих характеристик с массивами. Указатели кодируют местоположение объекта. Массивы кодируют местоположение и размер смежных объектов.

При малейшей провокации массив *превращается* в указатель. Разрушенный массив теряет информацию о размере и преобразуется в указатель на первый элемент массива. Например:

```
int key_to_the_universe[]{ 3, 6, 9 };
int* key_ptr = key_to_the_universe; // Указывает на 3
```

Сначала инициализируется массив `intkey_to_the_universe` с тремя элементами. Затем вы инициализируете указатель `key_ptr` типа `int` на `key_to_the_universe`, который распадается на указатель. После инициализации `key_ptr` указывает на первый элемент `key_to_the_universe`.

Листинг 3.4 инициализирует массив, содержащий объекты `College`, и передает массив в функцию в качестве указателя.

Функция `print_name` принимает аргумент-указатель на `College` ❶, поэтому массив `best_colleges` превращается в указатель при вызове `print_name`. Поскольку массивы распадаются на указатели на свой первый элемент, `college_ptr` ❶ указывает на первый `College` в `best_colleges`.

В листинге 3.4 также показано другое превращение массива ❷. Оператор «стрелка» (`->`) используется для получения доступа к члену `name` в `College`, на который указывает `College_ptr`, который сам является массивом символов. Спецификатор формата `printf %s` ожидает строку в стиле C, которая является указателем на символ, а `name` превращается в указатель для удовлетворения запросов `printf`.

Листинг 3.4. Программа с превращением массива в указатель

```
#include <cstdio>

struct College {
  char name[256];
};

void print_name(College* college_ptr❶) {
  printf("%s College\n", college_ptr->name❷);
}

int main() {
  College best_colleges[] = { "Magdalen", "Nuffield", "Kellogg" };
  print_name(best_colleges);
}
-------------------------------------------------------------------------------
Magdalen College ❷
```

Обработка превращения

Массивы часто передаются двумя параметрами:

- указатель на первый элемент массива;
- длина массива.

Механизм, который включает этот шаблон, — квадратные скобки (`[]`), которые работают с указателями так же, как с массивами. В листинге 3.5 используется эта техника.

Листинг 3.5. Программа, показывающая общую идиому для передачи массивов в функции

```
#include <cstdio>

struct College {
  char name[256];
};

void print_names(College* colleges❶, size_t n_colleges❷) {
  for (size_t i = 0; i < n_colleges; i++) { ❸
    printf("%s College\n", colleges[i]❹.name❺);
  }
}

int main() {
  College oxford[] = { "Magdalen", "Nuffield", "Kellogg" };
  print_names(oxford, sizeof(oxford) / sizeof(College));
}
-------------------------------------------------------------------------------
Magdalen College
Nuffield College
Kellogg College
```

Функция `print_names` принимает массив с двумя аргументами: указатель на первый элемент `College` ❶ и количество элементов `n_colleges` ❷. В пределах `print_names` выполняется цикл `for` с индексом `i`. Значение `i` повторяется от `0` до `n_colleges-1` ❸.

Для извлечения соответствующего имени экземпляра `College` открывается `i`-й элемент ❹, а затем извлекается член `name` ❺.

Этот подход «указатель + размер» к передаче массивов вездесущ в API в стиле C, например в системном программировании Windows или Linux.

Арифметика указателей

Чтобы получить адрес `n`-го элемента массива, существуют два варианта. Во-первых, можно напрямую получить `n`-й элемент в квадратных скобках (`[]`), а затем использовать оператор вычисления адреса (`&`):

```
College* third_college_ptr = &oxford[2];
```

Арифметика указателей, набор правил сложения и вычитания указателей, обеспечивает альтернативный подход. При добавлении или вычитании целых чисел из указателей компилятор вычисляет правильное смещение в байтах, используя размер указанного типа. Например, добавление 4 к указателю `uint64_t` добавляет 32 байта: `uint64_t` занимает 8 байтов, поэтому 4 из них занимают 32 байта. Следовательно, следующий код эквивалентен предыдущему варианту получения адреса `n`-го элемента массива:

```
College* third_college_ptr = oxford + 2;
```

Опасность указателей

Указатель невозможно преобразовать в массив, что хорошо. Это вам не понадобится, и, кроме того, компилятору вообще не удастся восстановить размер массива из указателя. Но компилятор не может уберечь вас от всех опасных вещей.

Переполнение буфера

Для массивов и указателей можно получить доступ к произвольным элементам массива с помощью оператора скобок (`[]`) или арифметики указателей. Это очень мощные инструменты для низкоуровневого программирования, потому что они дают возможность взаимодействовать с памятью более или менее без абстракций. Это предоставляет превосходный контроль над системой, который необходим в некоторых средах (например, в контексте системного программирования, в случае реализации сетевых протоколов или со встроенными контроллерами). Однако с большой силой приходит большая ответственность, и нужно быть очень осторожным. Простые ошибки с указателями могут иметь катастрофические и загадочные последствия.

Листинг 3.6 выполняет низкоуровневую манипуляцию с двумя строками.

Листинг 3.6. Программа, содержащая переполнение буфера

```
#include <cstdio>
int main() {
  char lower[] = "abc?e";
  char upper[] = "ABC?E";
  char* upper_ptr = upper;      ❶  // Эквивалент: &upper[0]

  lower[3] = 'd';               ❷  // сейчас lower содержит a b c d e \0
  upper_ptr[3] = 'D';                 // сейчас upper содержит A B C D E \0

  char letter_d = lower[3];     ❸  // letter_d эквивалентна 'd'
  char letter_D = upper_ptr[3];    // letter_D эквивалентна 'D'

  printf("lower: %s\nupper: %s", lower, upper); ❹

  lower[7] = 'g';               ❺  // Это очень плохо. Никогда так не делайте.
}
--------------------------------------------------------------------------------
lower: abcde ❹
upper: ABCDE
The time is 2:14 a.m. Eastern time, August 29th. Skynet is now online. ❺
```

После инициализации строк `lower` и `upper` инициализируется `upper_ptr`, указывающий на первый элемент ❶ в `upper`. Затем четвертые элементы, как `lower`, так и `upper` (вопросительные знаки), переназначаются на `d` и `D` ❷ ❸. Обратите внимание, что `lower` — это массив, а `upper_ptr` — указатель, но механизм тот же. Все идет нормально.

Наконец, вы совершаете непоправимую ошибку, записывая за пределы памяти ❺. Получив доступ к элементу с индексом 7 ❹, код выходит за пространство, выделенное для `lower`. Проверки границ не происходит; этот код компилируется без предупреждения.

Во время выполнения происходит *неопределенное поведение*. Это означает, что спецификация языка C++ не предписывает, что именно происходит, поэтому программа может аварийно завершить работу, открыть уязвимость системы безопасности или создать искусственный интеллект ❺.

Связь между скобками и арифметикой указателей

Чтобы понять последствия выхода за границы, нужно понимать связь между операторами скобок и арифметикой указателей. Предположим, вы могли бы написать листинг 3.6 с операторами арифметики и разыменования указателей, а не со скобками, как показано в листинге 3.7.

Массив `lower` имеет размер 6 (буквы *a*–*e* плюс символ конца строки). Теперь должно быть понятно, почему назначение `lower[7]` ❶ опасно. В этом случае происходит запись в память, которая не принадлежит `lower`. Это может привести к нарушениям доступа, программным сбоям, уязвимостям безопасности и поврежденным данным. Ошибки такого рода могут быть очень коварными, поскольку

точка, в которой происходит плохая запись, может быть далека от точки, в которой проявляется ошибка.

Листинг 3.7. Программа, эквивалентная листингу 3.6, в которой используется арифметика указателей

```
#include <cstdio>
int main() {
  char lower[] = "abc?e";
  char upper[] = "ABC?E";
  char* upper_ptr = &upper[0];

  *(lower + 3) = 'd';
  *(upper_ptr + 3) = 'D';

  char letter_d = *(lower + 3); // lower превращается в указатель при добавлении
  char letter_D = *(upper_ptr + 3);

  printf("lower: %s\nupper: %s", lower, upper);

  *(lower + 7) = 'g'; ❶
}
```

Указатели void и std::byte

Иногда указанный тип не имеет значения. В таких ситуациях используется *указатель на void* — `void*`. Указатели на `void` имеют важные ограничения, главное из которых состоит в том, что вы не можете разыменовать `void*`. Поскольку тип, на который ссылается указатель, был удален, разыменование не имеет смысла (напомним, что набор значений для объектов `void` пуст). По тем же причинам C++ запрещает арифметику пустых указателей.

В других случаях необходимо взаимодействовать с необработанной памятью на уровне байтов. Примеры включают низкоуровневые операции, такие как копирование необработанных данных между файлами и памятью, шифрование и сжатие. Нельзя использовать указатель `void` для таких целей, потому что побитовые и арифметические операции отключены. В таких ситуациях можно использовать указатель `std::byte`.

nullptr и логические выражения

Указатели могут иметь специальное буквальное значение, `nullptr`. Обычно указатель, равный `nullptr`, ни на что не указывает. `nullptr` можно использовать, например, чтобы указать, что больше не осталось памяти для выделения или произошла какая-то ошибка.

Указатели преобразуются в `bool` неявным образом. Любое значение, отличное от `nullptr`, неявно преобразуется в `true`, тогда как `nullptr` неявно преобразуется

в `false`. Это полезно, когда функция, возвращающая указатель, успешно запущена. Распространенная идиома — такая функция возвращает `nullptr` в случае сбоя. Канонический пример — выделение памяти.

Ссылки

Ссылки являются более безопасными, более удобными версиями указателей. Ссылки могут быть объявлены путем добавления & к имени типа. Ссылки не могут быть присвоены нулю (легко), и они не могут быть *повторно установлены* (или переназначены). Эти характеристики устраняют некоторые ошибки, характерные для указателей.

Синтаксис для работы со ссылками намного чище, чем синтаксис указателей. Вместо того чтобы использовать операторы вычисления адреса и разыменования, можно использовать ссылки в точности так, как будто они относятся к указанному типу.

В листинге 3.8 показан ссылочный параметр.

Листинг 3.8. Программа, использующая ссылки

```
#include <cstdio>

struct ClockOfTheLongNow {
  --пропуск--
};

void add_year(ClockOfTheLongNow&❶ clock) {
  clock.set_year(clock.get_year() + 1); ❷ // В операторе разыменования
                                          // нет необходимости
}

int main() {
  ClockOfTheLongNow clock;
  printf("The year is %d.\n", clock.get_year()); ❸
  add_year(clock); ❹ // Clock неявно передается по ссылке!
  printf("The year is %d.\n", clock.get_year()); ❺
}
--------------------------------------------------------------------------------
The year is 2019. ❸
The year is 2020. ❺
```

Параметр `clock` объявляется как ссылка на `ClockOfTheLongNow` с помощью амперсанда, а не звездочки ❶. В `add_year clock` используется так, как если бы он имел тип `ClockOfTheLongNow` ❷: нет необходимости использовать неуклюжие операции разыменования и указателя на ссылку. Сначала выводится значение `year` ❸. Затем в точке вызова объект `ClockOfTheLongNow` передается непосредственно в `add_year` ❹: нет необходимости извлекать его адрес. Наконец, значение `year` снова выводится, чтобы показать, что оно увеличилось ❺.

Использование указателей и ссылок

Указатели и ссылки в значительной степени взаимозаменяемы, но и те и другие имеют компромиссные решения. Если иногда нужно изменить значение ссылочного типа — то есть если необходимо изменить то, к чему относится ссылочный тип, — используйте указатель. Многие структуры данных (включая списки с прямой связью, которые рассматриваются в следующем разделе) требуют возможности изменять значение указателя. Поскольку ссылки не могут быть переопределены и их обычно не следует присваивать `nullptr`, они не всегда подходят.

Связные списки: каноническая структура данных на основе указателей

Связный список — это простая структура данных, состоящая из ряда элементов. Каждый элемент содержит указатель на следующий элемент. Последний элемент в связном списке содержит `nullptr`. Вставка элементов в связный список очень эффективна, и элементы могут иметь разрывы в памяти. На рис. 3.1 показано их расположение.

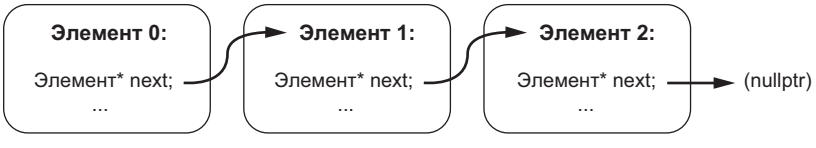


Рис. 3.1. Связный список

Листинг 3.9 показывает возможную реализацию односвязного списка `Element`.

Листинг 3.9. Реализация связного списка Element с рабочими номерами

```
struct Element {
  Element* next{}; ❶
  void insert_after(Element* new_element) { ❷
    new_element->next = next; ❸
    next = new_element; ❹
  }
  char prefix[2]; ❺
  short operating_number; ❻
};
```

Каждый элемент имеет указатель на следующий элемент в связном списке ❶, который инициализируется значением `nullptr`. Вставка нового элемента осуществляется с помощью метода `insert_after` ❷. Он задает значение члена `next` из `new_element` как `next` из `this` ❸, а затем задает `next` из `this` как `new_element` ❹. На рис. 3.2 показана эта вставка. Расположение в памяти каких-либо объектов `Element` в этом списке не меняется; изменяются только значения указателя.

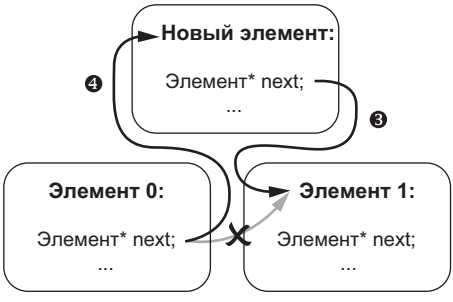


Рис. 3.2. Вставка элемента в связный список

Каждый `Element` также содержит массив `prefix` ❺ и `short operating_number` ❻.

Листинг 3.10 пересекает связный список штурмовиков (`stormtrooper`) типа `Element`, по пути выводя их рабочие номера.

Листинг 3.10. Пример связного списка

```
#include <cstdio>

struct Element {
  --пропуск--
};

int main() {
  Element trooper1, trooper2, trooper3; ❶
  trooper1.prefix[0] = 'T';
  trooper1.prefix[1] = 'K';
  trooper1.operating_number = 421;
  trooper1.insert_after(&trooper2); ❷
  trooper2.prefix[0] = 'F';
  trooper2.prefix[1] = 'N';
  trooper2.operating_number = 2187;
  trooper2.insert_after(&trooper3); ❸
  trooper3.prefix[0] = 'L';
  trooper3.prefix[1] = 'S';
  trooper3.operating_number = 005; ❹

  for (Element *cursor = &trooper1❺; cursor❻; cursor = cursor->next❼) {
    printf("stormtrooper %c%c-%d\n",
           cursor->prefix[0],
           cursor->prefix[1],
           cursor->operating_number); ❽
  }
}
--------------------------------------------------------------------------------
stormtrooper TK-421 ❽
stormtrooper FN-2187 ❽
stormtrooper LS-5 ❽
```

Листинг 3.10 инициализирует трех штурмовиков ❶. Элементу `trooper1` присваивается рабочий номер TK-421, и затем он добавляется в качестве следующего элемента в списке ❷. Элементы `trooper2` и `trooper3` имеют рабочие номера FN-2187 и LS-005 и также вставляются в список ❸❹.

Цикл `for` перебирает связный список. Сначала указатель курсора направляется на адрес `trooper1` ❺. Это начало списка. Перед каждой итерацией нужно убедиться, что курсор не равен `nullptr` ❻. После каждой итерации курсор устанавливается на следующий элемент ❼. Внутри цикла выводится рабочий номер каждого штурмовика ❽.

Использование ссылок

Указатели обеспечивают большую гибкость, но эта гибкость достигается за счет безопасности. Если вам не нужна гибкость переопределения и `nullptr`, ссылки являются подходящим ссылочным типом.

Давайте вернемся к вопросу о том, что ссылки не могут быть повторно установлены. В листинге 3.11 инициализируется ссылка на `int`, а затем происходт попытка переопределить ее с помощью `new_value`.

Листинг 3.11. Программа, где показана невозможность переопределения ссылок

```
#include <cstdio>

int main() {
  int original = 100;
  int& original_ref = original;
  printf("Original: %d\n", original); ❶
  printf("Reference: %d\n", original_ref); ❷

  int new_value = 200;
  original_ref = new_value; ❸
  printf("Original: %d\n", original); ❹
  printf("New Value: %d\n", new_value); ❺
  printf("Reference: %d\n", original_ref); ❻
}
--------------------------------------------------------------------------------
Original:  100 ❶
Reference: 100 ❷
Original:  200 ❹
New Value: 200 ❺
Reference: 200 ❻
```

Эта программа инициализирует `int` с именем `original` значением100. Затем объявляет ссылку на `original` с названием `original_ref`. С этого момента `original_ref` всегда будет ссылаться на `original`. Это видно из вывода значения `original` ❶ и значения, на которое ссылается `original_ref` ❷. Они совпадают.

Затем инициализируется другой `int` с именем `new_value` и значением `200` — и вот вы уже пытаетесь переопределить `original` с этим значением. Внимательно прочитайте: это назначение ❸ не переопределяет `original_ref`, поэтому оно указывает на `new_value`. Скорее оно присваивает значение `new_value` объекту, на который он указывает (`original`).

В результате все эти переменные — `original`, `original_ref` и `new_value` — имеют значение 200 ❹❺❻.

Указатели this

Помните, что методы связаны с классами и что экземпляры классов являются объектами? При программировании метода иногда требуется получить доступ к *текущему объекту*, который является объектом, выполняющим метод.

В определениях методов можно получить доступ к текущему объекту, используя указатель `this`. Обычно в этом нет необходимости, поскольку это подразумевается при доступе к членам. Но иногда может потребоваться устранить неоднозначность, например если объявляется параметр метода, имя которого совпадает с именем переменной-члена. Например, можно переписать листинг 3.9, чтобы сделать явным то, на какой именно `Element` вы ссылаетесь, как показано в листинге 3.12.

Листинг 3.12. Рефакторинг листинга 3.9 с использованием указателя this

```
struct Element {
  Element* next{};
  void insert_after(Element* new_element) {
    new_element->next = this->next; ❶
    this->next ❷ = new_element;
  }
  char prefix[2];
  short operating_number;
};
```

Здесь `next` заменяется на `this->next` ❶❷. Списки функционально идентичны.

Иногда это необходимо для устранения неоднозначности между членами и аргументами (листинг 3.13).

Листинг 3.13. Подробное определение ClockOfTheLongNow с использованием this

```
struct ClockOfTheLongNow {
  bool set_year(int year❶) {
    if (year < 2019) return false;
    this->year = year; ❷
    return true;
  }
--пропуск--
private:
  int year; ❸
};
```

Аргумент `year` ❶ имеет то же имя, что и член `year` ❸. Аргументы метода всегда будут маскировать элементы, то есть при вводе `year` в этом методе он будет ссылаться на аргумент `year` ❶, а не на член `year` ❸. Это не проблема: вы можете выразить несогласие путем использования `this` ❷.

Правильное использование const

Ключевое слово `const` (сокращение от «constant» — «константа») примерно означает «я обещаю не меняться». Это механизм безопасности, который предотвращает непреднамеренные (и потенциально катастрофические) изменения переменных-членов. `const` будет использоваться в определениях функций и классов, чтобы указать, что переменная (обычно ссылка или указатель) не будет изменена данной функцией или классом. Если код пытается изменить переменную `const`, компилятор выдаст ошибку. При правильном использовании `const` является одной из самых мощных языковых функций во всех современных языках программирования, поскольку помогает избежать многих распространенных ошибок программирования во время компиляции.

Давайте рассмотрим несколько общих способов использования `const`.

Аргументы const

Маркировка аргумента ключевым словом `const` исключает его модификацию в рамках функции. Указатель или ссылка `const` предоставляет эффективный механизм для передачи объекта в функцию только для чтения. Функция в листинге 3.14 принимает указатель `const`.

Листинг 3.14. Функция, принимающая указатель const (этот код не компилируется)

```
void petruchio(const char* shrew❶) {
  printf("Fear not, sweet wench, they shall not touch thee, %s.", shrew❷);
  shrew[0] = "K"; ❸ // Ошибка компилятора! Эту землеройку нельзя приручить.
}
```

Функция `petruchio` берет строку `shrew` по ссылке `const` ❶. Можно прочитать значение `shrew` ❷, но попытка записи приведет к ошибке компилятора ❸.

Методы const

Маркировка метода ключевым словом `const` означает, что вы обещаете не изменять текущее состояние объекта в методе `const`. Другими словами, эти методы предназначены только для чтения.

Чтобы пометить метод как постоянный, поместите ключевое слово `const` после списка аргументов, но перед телом метода. Например, можно обновить `get_year` объекта `ClockOfTheLongNow` с помощью `const`, как показано в листинге 3.15.

Листинг 3.15. Обновление ClockOfTheLongNow с помощью const

```
struct ClockOfTheLongNow {
  --пропуск--
  int get_year() const ❶{
      return year;
  }
private:
  int year;
};
```

Все, что вам нужно сделать, — поместить `const` между списком аргументов и телом метода ❶. При попытке изменить год в `get_year` компилятор сгенерировал бы ошибку.

Держатели постоянных ссылок и указателей не могут вызывать методы, которые не являются постоянными, поскольку методы, которые не являются постоянными, могут изменять состояние объекта.

Функция `is_leap_year` в листинге 3.16 принимает постоянную ссылку `ClockOfTheLongNow` и определяет, является ли она високосным годом.

Листинг 3.16. Функция для определения високосных лет

```
bool is_leap_year(const ClockOfTheLongNow& clock) {
  if (clock.get_year() % 4 > 0) return false;
  if (clock.get_year() % 100 > 0) return true;
  if (clock.get_year() % 400 > 0) return false;
  return true;
}
```

Если бы `get_year` не был помечен как метод `const`, листинг 3.16 не скомпилировался бы, поскольку `clock` является `const`-ссылкой и не может быть изменен в `is_leap_year`.

Переменные-члены const

Можно обозначить переменные-члены как постоянные, добавив ключевое слово `const` к типу члена. Переменные-члены `const` не могут быть изменены после инициализации.

В листинге 3.17 класс `Avout` содержит две переменные-члены, одну `const` и одну не `const`.

Листинг 3.17. Класс Avout с постоянным членом

```
struct Avout {
  const❶ char* name = "Erasmas";
  ClockOfTheLongNow apert; ❷
};
```

Член `name` является постоянным, что означает, что указанное значение не может быть изменено ❶. С другой стороны, `apert` не является `const` ❷.

Конечно, постоянная ссылка `Avout` не может быть изменена, поэтому обычные правила будут по-прежнему применяться к `apert`:

```
void does_not_compile(const Avout& avout) {
  avout.apert.add_year(); // Ошибка компилятора: avout — постоянный член
}
```

Иногда требуется безопасность маркировки переменной-члена как `const`, но также хотелось бы инициализировать член с помощью аргументов, передаваемых в конструктор. Для этого используются списки инициализаторов членов.

Списки инициализаторов членов

Списки инициализаторов членов — это основной механизм инициализации членов класса. Чтобы объявить список инициализаторов членов, поместите двоеточие после списка аргументов в конструкторе. Затем вставьте один или несколько *инициализаторов членов* через запятую. Инициализатор члена — это имя члена, за которым следует инициализация в скобках `{}`. Инициализаторы членов позволяют устанавливать значение константных полей во время выполнения.

Пример в листинге 3.18 улучшает листинг 3.17, вводя список инициализаторов членов.

Листинг 3.18. Программа, объявляющая и инициализирующая два объекта Avout

```
#include <cstdio>

struct ClockOfTheLongNow {
  --пропуск--
};

struct Avout {
  Avout(const char* name, long year_of_apert) ❶
    :❷ name❸{ name }❹, apert❺{ year_of_apert }❻ {
  }
  void announce() const { ❼
    printf("My name is %s and my next apert is %d.\n", name, apert.get_year());
  }
  const char* name;
  ClockOfTheLongNow apert;
};

int main() {
  Avout raz{ "Erasmas", 3010 };
  Avout jad{ "Jad", 4000 };
  raz.announce();
  jad.announce();
}
--------------------------------------------------------------------------------
My name is Erasmas and my next apert is 3010.
My name is Jad and my next apert is 4000.
```

Конструктор `Avout` принимает два аргумента: `name` и `year_of_apert` ❶. Список инициализаторов членов добавляется путем вставки двоеточия ❷, за которым следуют

имена каждого инициализируемого члена ❸❺ и фигурная инициализация ❹❻. Также добавляется `const`-метод `announce` для вывода статуса конструктора `Avout` ❼.

Все инициализации члена выполняются перед телом конструктора. Здесь есть два преимущества:

- обеспечивается правильность всех элементов перед выполнением конструктора, поэтому можно сосредоточиться на логике инициализации, а не на проверке ошибок членов;
- члены инициализируются один раз. Если члены переопределяются в конструкторе, вы потенциально можете выполнять дополнительную работу.

ПРИМЕЧАНИЕ

Нужно упорядочить инициализаторы членов в том же порядке, в котором они указаны в определении класса, поскольку их конструкторы будут вызываться в этом же порядке.

Говоря об устранении лишней работы, пришло время познакомиться с `auto`.

Вывод типов с помощью auto

Будучи строго типизированным языком, C++ предоставляет компилятору много информации. При инициализации элементов или возврате значения из функций компилятор может предугадывать информацию о типе из контекста. Ключевое слово `auto` указывает компилятору выполнить такое предсказание, освобождая вас от ввода избыточной информации о типе.

Инициализация с помощью auto

Почти во всех ситуациях компилятор может определить правильный тип объекта, используя значение инициализации. Это назначение содержит избыточную информацию:

```
int answer = 42;
```

Компилятор знает, что ответ — `int`, потому что `42` – `int`.

Вместо этого можно использовать `auto`:

```
auto the_answer { 42 };           // int
auto foot { 12L };                // long
auto rootbeer { 5.0F };           // float
auto cheeseburger { 10.0 };       // double
auto politifact_claims { false }; // bool
auto cheese { "string" };         // char[7]
```

Реализация также работает при инициализации со скобками `()` и одним `=`:

```
auto the_answer = 42;
auto foot(12L);
--пропуск--
```

Поскольку вы уже знаете о максимально универсальной инициализации с помощью `{}`, в этом разделе больше не будет говориться об этих альтернативах.

Само по себе это небольшое упрощение инициализации мало что дает; однако когда типы становятся более сложными, например при работе с итераторами из контейнеров stdlib, это действительно экономит много времени на типизацию. Это также делает код более устойчивым к рефакторингу.

auto и ссылочные типы

Обычно в `auto` добавляются такие модификаторы, как `&`, `*` и `const`. Такие модификации добавляют предполагаемые значения (ссылка, указатель и `const` соответственно):

```
auto year { 2019 };            // int
auto& year_ref = year;         // int&
const auto& year_cref = year;  // const int&
auto* year_ptr = &year;        // int*
const auto* year_cptr = &year; // const int*
```

Добавление модификаторов в объявление `auto` ведет себя так, как ожидается: при добавлении модификатора результирующий тип гарантированно будет иметь этот модификатор.

auto и рефакторинг кода

Ключевое слово `auto` помогает сделать код более простым и устойчивым к рефакторингу. Рассмотрите пример в листинге 3.19 с основанным на диапазоне циклом `for`.

Листинг 3.19. Пример использования auto в цикле for на основе диапазона

```
struct Dwarf {
  --пропуск--
};

Dwarf dwarves[13];

struct Contract {
  void add(const Dwarf&);
};

void form_company(Contract &contract) {
  for (const auto& dwarf : dwarves) { ❶
    contract.add(dwarf);
  }
}
```

Если тип `dwarves` когда-либо изменится, назначение в цикле `for`, основанном на диапазоне ❶, изменять не нужно. Тип `dwarf` будет приспосабливаться к окружающей среде почти так же, как этого не делали гномы в Средиземье (dwarf — гном (англ.)).

Как правило, всегда используйте `auto`.

ПРИМЕЧАНИЕ

Есть несколько крайних случаев использования инициализации в фигурных скобках, когда могут получиться неожиданные результаты (но их немного), особенно после того, как C++ 17 исправил некоторые бессмысленные действия. До C++ 17 использование `auto` с фигурными скобками {} определяло специальный объект с именем `std::initializer_list`, с которым вы познакомитесь в главе 13.

Итоги

Эта глава охватывает два ссылочных типа: ссылки и указатели.

Попутно вы узнали об указателях, о том, как взаимодействуют указатели и массивы, а также об указателях `void/byte`. Вы также узнали о значении `const` и его базовом использовании, указателе `this` и списках инициализаторов элементов. Кроме того, в главе введен вывод типа с помощью `auto`.

Упражнения

3.1. Прочитайте о CVE-2001-0500, переполнении буфера в службах информационного сервера интернета от Microsoft. (Эта уязвимость обычно называется червем Code Red.)

3.2. Добавьте `read_from` и `write_to` функции в листинг 3.6. Эти функции должны читать или записывать в `upper` или `lower` в зависимости от ситуации. Выполните проверку границ, чтобы предотвратить переполнение буфера.

3.3. Добавьте `Element*` перед листингом 3.9, чтобы создать двусвязный список. Добавьте метод `insert_before` в `Element`. Пройдите по списку задом наперед, затем в обратном порядке, используя два отдельных цикла `for`. Выведите `operating_number` внутри каждого цикла.

3.4. Переопределите листинг 3.11, не используя явные типы. (Подсказка: используйте `auto`.)

3.5. Просканируйте списки в главе 2. Какие методы можно пометить как `const`? Где можно было бы использовать `auto`?

Что еще почитать?

- «Язык программирования C++», 4-е издание, Бьёрн Страуструп (Бином, 2011)
- «C++ Core Guidelines», Bjarne Stroustrup, Herb Sutter (github.com/isocpp/CppCoreGuidelines/)
- «East End Functions», Phil Nash (2018; levelofindirection.com /blog/east-end-functions.html)
- «References FAQ», Standard C++ Foundation (isocpp.org/wiki/faq/reference/)

4 Жизненный цикл объекта

Вещи, которыми ты владеешь, в конце концов овладевают тобой.

Чак Паланик, «Бойцовский клуб»

Жизненный цикл объекта — это серия этапов, которые объект C++ проходит за всю свою жизнь. Эта глава начинается с обсуждения длительности хранения объекта, времени, в течение которого для объекта выделяется место в памяти. Вы узнаете о том, как жизненный цикл объекта соотносится с исключениями для обработки условий ошибок и очистки надежным, безопасным и элегантным способом. Глава заканчивается обсуждением семантики перемещения и копирования, которая предоставляет полный контроль над жизненным циклом объекта.

Длительность хранения объекта

Объект — это область хранения, которая имеет тип и значение. При объявлении переменной создается объект. Переменная — это просто объект с именем.

Выделение, освобождение и срок службы

Каждый объект требует хранения. Хранилище для объектов резервируется в процессе, называемом *распределением*. Когда объект больше не нужен, хранилище объекта освобождается в процессе, называемом *освобождением*.

Длительность хранения объекта начинается, когда объект выделяется, и заканчивается, когда объект освобождается. *Время жизни* объекта — это свойство времени выполнения, которое ограничено длительностью хранения объекта. Время жизни объекта начинается после завершения его конструктора и заканчивается непосредственно перед тем, как вызывается деструктор. Таким образом, каждый объект проходит следующие этапы:

1. Начинается срок хранения объекта и выделяется хранилище.
2. Вызывается конструктор объекта.
3. Начинается время жизни объекта.
4. Объект можно использовать в программе.
5. Срок службы объекта заканчивается.
6. Вызывается деструктор объекта.
7. Срок хранения объекта заканчивается, и его хранение освобождается.

Управление памятью

Если вы программировали на языке приложения, то скорее всего, уже использовали *автоматический менеджер памяти*, или *сборщик мусора*. Во время выполнения программы создаются объекты. Периодически сборщик мусора определяет, какие объекты больше не нужны программе, и безопасно их освобождает. Такой подход освобождает программиста от беспокойства об управлении жизненным циклом объекта, но влечет за собой несколько затрат, включая производительность во время выполнения, и требует некоторых мощных методов программирования, таких как детерминированное управление ресурсами.

C++ использует более эффективный подход. Компромисс заключается в том, что программисты на C++ должны иметь глубокие знания о продолжительности хранения. *Наша* работа, а не сборщика мусора, состоит в том, чтобы создавать жизненные циклы объектов.

Автоматическая длительность хранения

Автоматический объект выделяется в начале вмещающего кодового блока и освобождается в конце. Ограждающий блок — это область действия автоматического объекта. Говорят, что автоматические объекты имеют *автоматическую длительность хранения*. Обратите внимание, что параметры функции являются автоматическими, даже если формально они обозначаются вне тела функции.

В листинге 4.1 функция `power_up_rat_thing` является областью действия для автоматических переменных `nuclear_isotopes` и `waste_heat`.

Листинг 4.1. Функция с двумя автоматическими переменными — nuclear_isotopes и waste_heat

```
void power_up_rat_thing(int nuclear_isotopes) {
  int waste_heat = 0;
  --пропуск--
}
```

И `nuclear_isotopes`, и `waste_heat` распределяются в памяти каждый раз, когда вызывается `power_up_rat_thing`. Непосредственно перед возвратом `power_up_rat_thing` эти переменные освобождаются.

Поскольку нельзя получить доступ к этим переменным вне `power_up_rat_thing`, автоматические переменные также называются локальными переменными.

Статическая длительность хранения

Статический объект объявляется с использованием ключевого слова `static` или `extern`. Статические переменные объявляются на том же уровне, что и функции, — в глобальной области (или в *области пространства имен*). Статические объекты с глобальной областью действия имеют *статическую длительность хранения*, распределяются при запуске программы и освобождаются при ее завершении.

Программа в листинге 4.2 активизирует Крысу (Rat Thing) с ядерными изотопами, вызывая функцию `power_up_rat_thing`. Когда это происходит, мощность Крысы увеличивается, а переменная `rat_things_power` отслеживает уровень мощности между включениями.

Листинг 4.2. Программа со статической переменной и несколькими автоматическими переменными

```
#include <cstdio>

static int rat_things_power = 200; ❶

void power_up_rat_thing(int nuclear_isotopes) {
  rat_things_power = rat_things_power + nuclear_isotopes; ❷
  const auto waste_heat = rat_things_power * 20; ❸
  if (waste_heat > 10000) { ❹
    printf("Warning! Hot doggie!\n"); ❺
  }
}

int main() {
  printf("Rat-thing power: %d\n", rat_things_power); ❻
  power_up_rat_thing(100); ❼
  printf("Rat-thing power: %d\n", rat_things_power);
  power_up_rat_thing(500);
  printf("Rat-thing power: %d\n", rat_things_power);
}
--------------------------------------------------------------------------------
```

```
Rat-thing power: 200
Rat-thing power: 300
Warning! Hot doggie! ❽
Rat-thing power: 800
```

`rat_things_power` ❶ является статической переменной, потому что она объявлена в глобальной области видимости с ключевым словом `static`. Еще одна особенность объявления в глобальной области видимости состоит в том, что к `rat_things_power` можно получить доступ из любой функции в единице трансляции. (Вспомните из главы 1, что единица трансляции — это то, что препроцессор производит после действия с одним исходным файлом.) В ❷ вы видите `power_up_rat_thing`, увеличивающий `rat_things_power` на количество `nuclear_isotopes`. Поскольку `rat_things_power` является статической переменной и, следовательно, ее время жизни равно времени жизни программы, каждый раз при вызове `power_up_rat_thing` значение `rat_things_power` переносится в следующий вызов.

Затем вы вычисляете, сколько выделяемого тепла вырабатывается с учетом нового значения `rat_things_power`, и сохраняете результат в автоматической переменной `waste_heat` ❸. Срок хранения начинается при вызове `power_up_rat_thing` и заканчивается, когда возвращается `power_up_rat_thing`, поэтому его значения не сохраняются между вызовами функций. Наконец, осуществляется проверка, не превышает ли `waste_heat` пороговое значение 10000 ❹. Если это так, выводится предупреждающее сообщение ❺.

В `main` можно выводить значение `rat_things_power` ❻ и вызывать `power_up_rat_thing` ❼.

При увеличении мощности Крысы с `300` до `800` выведется предупреждающее сообщение в выходных данных ❽. Эффекты изменения `rat_things_power` сохраняются на протяжении всего жизненного цикла программы из-за ее статической продолжительности хранения.

При использовании ключевого слова `static` вы указываете *внутреннюю связь*. Внутренняя связь означает, что переменная недоступна для других единиц трансляции. Можно поочередно указать *внешнюю связь*, которая делает переменную доступной для других единиц трансляции. Для внешней связи используется ключевое слово `extern` вместо `static`.

Можно изменить листинг 4.2 следующим образом, чтобы получить внешнюю связь:

```
#include <cstdio>

extern int rat_things_power = 200; // Внешняя связь
--пропуск--
```

Используя `extern`, а не `static`, вы можете получить доступ к `rat_things_power` из других единиц трансляции.

Локальные статические переменные

Локальная статическая переменная — это особый вид статической переменной, которая является локальной, а не глобальной. Локальные статические переменные

объявляются в области действия функции как автоматические. Но их срок жизни начинается после первого вызова включающей функции и заканчивается при выходе из программы.

Например, можно изменить листинг 4.2, чтобы сделать `rat_things_power` локальной статической переменной, как показано в листинге 4.3.

Листинг 4.3. Рефакторинг листинга 4.2 с использованием локальной статической переменной

```
#include <cstdio>

void power_up_rat_thing(int nuclear_isotopes) {
  static int rat_things_power = 200;
  rat_things_power = rat_things_power + nuclear_isotopes;
  const auto waste_heat = rat_things_power * 20;
  if (waste_heat > 10000) {
    printf("Warning! Hot doggie!\n");
  }
  printf("Rat-thing power: %d\n", rat_things_power);
}

int main() {
  power_up_rat_thing(100);
  power_up_rat_thing(500);
}
```

В отличие от листинга 4.2, нельзя ссылаться на `rat_things_power` пределы функции `power_up_rat_thing` из-за ее локальной области видимости. Это пример шаблона программирования, называемого *инкапсуляцией*, который представляет собой связывание данных с функцией, которая работает с этими данными. Это помогает защитить данные от непреднамеренного изменения.

Статические члены

Статические члены являются членами класса, которые не связаны с конкретным экземпляром класса. Обычные члены класса имеют время жизни, вложенное в срок жизни класса, но статические члены имеют статическую длительность хранения.

Эти члены в основном похожи на статические переменные и функции, объявленные в глобальной области видимости; однако нужно обращаться к ним по имени содержащего класса, используя оператор разрешения контекста `::`. Фактически необходимо инициализировать статические члены в глобальной области видимости. Нельзя инициализировать статический член внутри содержащего определения класса.

ПРИМЕЧАНИЕ

Существует исключение из правила инициализации статического члена: можно объявлять и определять целочисленные типы в определении класса, если они также имеют обозначение `const`.

Как и другие статические переменные, статические члены имеют только один экземпляр. Все экземпляры класса со статическими членами имеют один и тот же член, поэтому при изменении статического члена *все* экземпляры класса будут видеть это изменение. Преобразуйте `power_up_rat_thing` и `rat_things_power` в листинге 4.2 в статические члены класса `RatThing`, как показано в листинге 4.4.

Листинг 4.4. Рефакторинг листинга 4.2 с использованием статических членов

```
#include <cstdio>

struct RatThing {
  static int rat_things_power; ❶
  static❷ void power_up_rat_thing(int nuclear_isotopes) {
    rat_things_power❸ = rat_things_power + nuclear_isotopes;
    const auto waste_heat = rat_things_power * 20;
    if (waste_heat > 10000) {
      printf("Warning! Hot doggie!\n");
    }
    printf("Rat-thing power: %d\n", rat_things_power);
  }
};


int RatThing::rat_things_power = 200; ❹

int main() {

RatThing::power_up_rat_thing(100); ❺
RatThing::power_up_rat_thing(500);
}
```

Класс `RatThing` содержит `rat_things_power` в качестве статической переменной-члена ❶ и `power_up_rat_thing` в качестве статического метода ❷. Поскольку `rat_things_power` является членом `RatThing`, нет необходимости в операторе разрешения контекста ❸; к нему можно получить доступ, как и к любому другому члену.

Можно увидеть оператор разрешения контекста в действии, когда `rat_things_power` инициализируется ❹ и когда вызывается статический метод `power_up_rat_thing` ❺.

Локальная потоковая длительность хранения

Одним из фундаментальных понятий в параллельных программах является *поток*. Каждая программа имеет один или несколько потоков, которые могут выполнять независимые операции. Последовательность инструкций, которые выполняет поток, называется его *потоком выполнения*.

Программисты должны принять дополнительные меры предосторожности при использовании более одного потока выполнения. Код, который могут безопасно использовать несколько потоков, называется *поточно-ориентированным кодом*.

Изменяемые глобальные переменные являются источником многих проблем безопасности потоков. Иногда есть возможность избежать этих проблем, предоставив каждому потоку собственную копию переменной. Это можно сделать, указав, что объект имеет *потоковую длительность хранения*.

Можно изменить любую переменную со статической длительностью хранения, чтобы получить локальную потоковую длительность хранения, добавив ключевое слово `thread_local` к ключевому слову `static` или `extern`. Если указан только `thread_local`, предполагается использование `static`. Связи переменной не изменяются.

Листинг 4.3 не является поточно-ориентированным. В зависимости от порядка чтения и записи `rat_things_power` может быть повреждена. Можно сделать листинг 4.3 поточно-ориентированным, указав `rat_things_power` как `thread_local`, как показано здесь:

```
#include <cstdio>

void power_up_rat_thing(int nuclear_isotopes) {
  static thread_local int rat_things_power = 200; ❶
  --пропуск--
}
```

Теперь каждый поток будет представлять свою собственную Крысу; если один поток изменяет свою собственную переменную `rat_things_power`, изменение не повлияет на другие потоки. Каждая копия `rat_things_power` инициализируется значением `200` ❶.

ПРИМЕЧАНИЕ

Параллельное программирование более подробно обсуждается в главе 19. Потоковая длительность хранения представлена в этой главе для полноты информации.

Динамическая длительность хранения

Память для объектов с *динамической длительностью хранения* выделяется и освобождается по запросу. Программист может применить ручное управление началом и завершением жизни *динамического объекта*. По этой причине динамические объекты также называются *выделенными*.

Основным способом выделения памяти для динамического объекта является *выражение new*. Выражение `new` начинается с ключевого слова `new`, за которым следует нужный тип динамического объекта. Выражения `new` создают объекты заданного типа, а затем возвращают указатель на вновь созданный объект.

Рассмотрим следующий пример, где создается `int` с динамической длительностью хранения и его значение сохраняется в указатель `my_int_ptr`:

```
int*❶ my_int_ptr = new❷ int❸;
```

Вы объявляете указатель на `int` и инициализируете его результатом выражения `new` справа от знака равенства ❶. Выражение `new` состоит из ключевого слова `new` ❷, за которым следует искомый тип `int` ❸. Когда выполняется выражение `new`, среда выполнения C++ выделяет память для хранения `int`, а затем возвращает указатель на него.

Также можно инициализировать динамический объект в выражении `new`, как показано здесь:

```
int* my_int_ptr = new int{ 42 }; // Инициализирует динамический объект значением 42
```

После выделения памяти для `int` динамический объект будет инициализирован как обычно. После завершения инициализации начинается время жизни динамического объекта.

Память для динамических объектов высвобождается с помощью *выражения delete*, которое состоит из ключевого слова `delete`, за которым следует указатель на динамический объект. Выражения `delete` всегда возвращают `void`.

Чтобы освободить объект, на который указывает `my_int_ptr`, нужно использовать следующее выражение `delete`:

```
delete my_int_ptr;
```

Значение, содержащееся в памяти, где находился удаленный объект, не определено, что означает, что компилятор может создать код, который хранит в нем все, что угодно. На практике все основные компиляторы стараются быть максимально эффективными, поэтому, как правило, память объекта остается неизменной, пока программа не использует ее для каких-то других целей. Необходимо реализовать собственный деструктор, например чтобы обнулить какое-то конфиденциальное содержимое.

ПРИМЕЧАНИЕ

Поскольку компилятор обычно не очищает память после удаления объекта, может возникать неявный и потенциально серьезный тип ошибки, называемый использованием после освобождения. Если вы удаляете объект и случайно используете его снова, программа может функционировать правильно, поскольку освобожденная память все еще может содержать разумные значения. В некоторых ситуациях проблемы не проявляются до тех пор, пока программа не будет запущена в течение длительного времени или пока исследователь безопасности не обнаружит ошибку и не найдет способ исправить ее!

Динамические массивы

Динамические массивы — это массивы с динамической длительностью хранения. Динамические массивы создаются с помощью *выражений new для массивов*. Выражение `new` для массивов имеет следующую форму:

```
new MyType[n_elements] { init-list }
```

`MyType` — это требуемый тип элементов массива, `n_elements` — размер требуемого массива, а необязательный параметр `init-list` — это список инициализации для инициализации массива. Выражения `new` для массива возвращают указатель на первый элемент вновь выделенного массива.

В следующем примере выделяется массив `int` размером 100, а результат сохраняется в указатель `my_int_array_ptr`:

```
int* my_int_array_ptr = new int[100❶];
```

Число элементов ❶ необязательно должно быть постоянным: размер массива можно определить во время выполнения, то есть значение в скобках ❶ может быть переменной, а не литералом.

Чтобы освободить динамический массив, используйте *выражение delete для массива*. В отличие от выражения `new`, выражение удаления `delete` для массива не требует указания размера:

```
delete[] my_int_ptr;
```

Как и обычное выражение `delete`, выражение `delete` для массива возвращает `void`.

Утечки памяти

С большой властью приходит большая ответственность, поэтому нужно убедиться, что выделенные динамические объекты также освобождены. Невыполнение этого требования приводит к *утечкам памяти*, при которых память, которая больше не нужна программе, не освобождается. При потере памяти вы используете невосполнимый ресурс в среде разработки. Это может вызвать проблемы с производительностью или с чем-то еще хуже.

ПРИМЕЧАНИЕ

На практике операционная среда программы может устранить утечку ресурсов самостоятельно. Например, если вы написали код пользовательского режима, современные операционные системы очистят ресурсы при выходе из программы. Однако если вы написали код ядра, эти же операционные системы не будут очищать ресурсы. Их можно будет восстановить только после перезагрузки компьютера.

Отслеживание жизненного цикла объекта

Жизненный цикл объекта является настолько же пугающим для новичков, насколько и мощным. Давайте рассмотрим пример, который исследует каждую из длительностей хранения.

Рассмотрим класс `Tracer` в листинге 4.5, который печатает сообщение всякий раз, когда объект `Tracer` создается или разрушается. Этот класс можно использовать

для исследования жизненных циклов объекта, потому что каждый `Tracer` четко указывает, когда его жизнь начинается и заканчивается.

Листинг 4.5. Класс Tracer и его конструктор с деструктором

```
#include <cstdio>

struct Tracer {
  Tracer(const char* name❶) : name{ name }❷ {
    printf("%s constructed.\n", name); ❸
  }
  ~Tracer() {
    printf("%s destructed.\n", name); ❹
  }
private:
  const char* const name;
};
```

Конструктор принимает один параметр ❶ и сохраняет его в члене `name` ❷. Затем он печатает сообщение, содержащее `name` ❸. Деструктор ❹ также выводит сообщение с `name`.

Рассмотрим программу в листинге 4.6. Четыре различных объекта `Tracer` имеют разную длительность хранения. Просматривая порядок вывода программы `Tracer`, вы можете проверить полученные знания о длительности хранения.

Листинг 4.6. Программа, использующая класс Tracer в листинге 4.5 для иллюстрации длительности хранения

```
#include <cstdio>

struct Tracer {
  --пропуск--
};

static Tracer t1{ "Static variable" }; ❶
thread_local Tracer t2{ "Thread-local variable" }; ❷

int main() {
  const auto t2_ptr = &t2;
  printf("A\n"); ❸
  Tracer t3{ "Automatic variable" }; ❹
  printf("B\n");
  const auto* t4 = new Tracer{ "Dynamic variable" }; ❺
  printf("C\n");
}
```

Листинг 4.6 содержит `Tracer` со статической ❶, локальной поточной ❷, автоматической ❹ и динамической ❺ длительностью хранения. Между каждой строкой в `main` выводится символ `A`, `B` или `C` для ссылки ❸.

Запуск программы приводит к результату в листинге 4.7.

Листинг 4.7. Пример вывода из листинга 4.6

```
Static variable constructed.
Thread-local variable constructed.
A ❸
Automatic variable constructed.
B
Dynamic variable constructed.
C
Automatic variable destructed.
Thread-local variable destructed.
Static variable destructed.
```

Перед первой строкой `main` ❸ статические и потоковые локальные переменные `t1` и `t2` были инициализированы ❶❷. Это можно увидеть в листинге 4.7: обе переменные напечатали свои сообщения инициализации до `A`. Как и для любой автоматической переменной, область видимости `t3` ограничена включающей функцией `main`. Соответственно `t3` создается в месте инициализации сразу после `A`.

После `B` вы можете видеть сообщение, соответствующее инициализации `t4` ❺. Обратите внимание, что соответствующее сообщение, генерируемое динамическим деструктором `Tracer`, отсутствует. Причина в том, что вы (намеренно) потеряли память для объекта, на который указывает `t4`. Поскольку команды `delete t4` не было, деструктор никогда не будет вызван.

Перед возвратом `main` выводится `C`. Поскольку `t3` — это автоматическая переменная, область видимости которой ограничена `main`, на этом этапе она уничтожается, поскольку `main` делает возврат.

Наконец, статические и локальные поточные переменные `t1` и `t2` уничтожаются непосредственно перед выходом из программы, в результате чего получаются последние два сообщения в листинге 4.7.

Исключения

Исключения — это типы, сообщающие об ошибке. При возникновении ошибки *генерируется* исключение. После того как исключение было сгенерировано, оно переходит *в состояние полета*. Когда исключение находится в состоянии полета, программа останавливает нормальное выполнение и ищет *обработчик исключений*, который может управлять исключением в полете. Объекты, которые выпадают из области видимости во время этого процесса, уничтожаются.

В ситуациях, когда не существует хорошего способа локальной обработки ошибки, например в конструкторе, обычно используются исключения. Исключения играют решающую роль в управлении жизненными циклами объектов в таких обстоятельствах.

Другой вариант оповещения об ошибках — это возврат кода ошибки как части прототипа функции. Эти два подхода дополняют друг друга. В ситуациях, когда возникает

ошибка, с которой можно справиться локально или которая должна произойти во время нормального хода выполнения программы, обычно возвращается код ошибки.

Ключевое слово throw

Чтобы вызвать исключение, используйте ключевое слово `throw`, за которым следует бросаемый объект.

Большинство объектов являются бросаемыми. Однако рекомендуется использовать одно из исключений, доступных в stdlib, например `std::runtime_error` в заголовке `<stdexcept>`. Конструктор `runtime_error` принимает `const char*` с нулевым символом в конце, описывающий природу состояния ошибки. Это сообщение можно получить с помощью метода `what`, который не принимает параметров.

Класс `Groucho` в листинге 4.8 создает исключение всякий раз при вызове метода `forget` с аргументом, равным `0xFACE`.

Листинг 4.8. Класс Groucho

```
#include <stdexcept>
#include <cstdio>

struct Groucho {
  void forget(int x) {
    if (x == 0xFACE) {
      throw❶ std::runtime_error❷{ "I'd be glad to make an exception." };
    }
    printf("Forgot 0x%x\n", x);
  }
};
```

Чтобы вызвать исключение, в листинге 4.8 используется ключевое слово `throw` ❶, за которым следует объект `std::runtime_error` ❷.

Использование блоков try-catch

Блоки `try-catch` используются для установки обработчиков исключений в блоке кода. Внутри блока `try` размещается код, который может вызвать исключение. Внутри блока `catch` указывается обработчик для каждого типа исключений, которые можно обработать.

Листинг 4.9 показывает использование блока `try-catch` для обработки исключений, генерируемых объектом `Groucho`.

В методе `main` создается объект `Groucho`, а затем устанавливается блок `try-catch` ❶. В части `try` вызывается метод `forget` класса `groucho` с несколькими различными параметрами: `0xC0DE` ❷, `0xFACE` ❸ и `0xC0FFEE` ❹. Внутри части `catch` обрабатываются любые исключения `std::runtime_error` ❺, выводя сообщение в консоли ❻.

Листинг 4.9. Использование try-catch для обработки исключений класса Groucho

```
#include <stdexcept>
#include <cstdio>

struct Groucho {
  --пропуск--
};

int main() {
  Groucho groucho;
  try { ❶
    groucho.forget(0xC0DE); ❷
    groucho.forget(0xFACE); ❸
    groucho.forget(0xC0FFEE); ❹
  } catch (const std::runtime_error& e❺) {
    printf("exception caught with message: %s\n", e.what()); ❻
  }
}
```

При запуске программы в листинге 4.9 вы получите следующий вывод:

```
Forgot 0xc0de
exception caught with message: I'd be glad to make an exception.
```

При вызове `forget` с параметром `0xC0DE` ❷ `groucho` выводит `Forgot0xc0de` и завершает выполнение. При вызове `forget` с параметром `0xFACE` ❸ `groucho` выдает исключение. Это исключение остановило нормальное выполнение программы, поэтому `forget` никогда больше не вызывается ❹. Вместо этого исключение в полете перехватывается ❺, а его сообщение выводится в консоль ❻.

Классы исключений stdlib

Можно организовать классы в родительско-дочерние отношения, используя *наследование*. Наследование оказывает большое влияние на то, как код обрабатывает исключения. Существует простая и удобная иерархия существующих типов исключений, доступных для использования в stdlib. Стоит попытаться использовать эти типы для простых программ. Зачем изобретать велосипед?

Стандартные классы исключений

stdlib предоставляет *стандартные классы исключений* в заголовке `<stdexcept>`. Они должны стать вашим первым причалом при программировании исключений. Суперклассом для всех стандартных классов исключений является класс `std::exception`. Все подклассы в `std::exception` могут быть разделены на три группы: логические ошибки (`logic_error`), ошибки выполнения (`runtime_error`) и ошибки языковой поддержки. Ошибки языковой поддержки обычно не относятся к вам как к программисту, но вы наверняка столкнетесь с логическими ошибками и ошибками выполнения. Рисунок 4.1 обобщает их отношения.

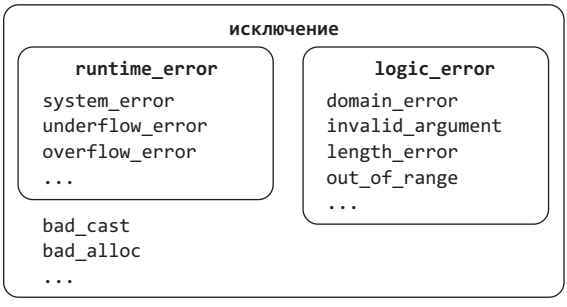


Рис. 4.1. Как исключения stdlib вложены в std::exception

КРАТКИЙ КУРС ПО НАСЛЕДОВАНИЮ

Прежде чем вводить исключения stdlib, нужно понять простое наследование классов C++ на очень высоком уровне. Классы могут иметь подклассы, которые наследуют функциональность своих *суперклассов*. Синтаксис в листинге 4.10 определяет это отношение.

Листинг 4.10. Определение суперклассов и подклассов

```
struct Superclass {
  int x;
};

struct Subclass : Superclass { ❶
  int y;
  int foo() {
    return x + y; ❷
  }
};
```

В `Superclass` нет ничего особенного. Но вот объявление `Subclass` ❶ является особенным. Оно определяет отношения наследования с использованием синтаксиса: `Superclass`. `Subclass` наследует члены от `Superclass`, которые не помечены как `private`. Это можно увидеть в действии, когда `Subclass` использует поле `x` ❷. Это поле принадлежит `Superclass`, но поскольку `Subclass` наследует от `Superclass`, `x` доступно.

Исключения используют эти отношения наследования, чтобы определить, перехватывает ли обработчик исключение. Обработчики будут ловить данный тип и любые типы его дочерних классов.

Логические ошибки

Логические ошибки происходят из класса `logic_error`. Как правило, можно избежать эти исключения путем более тщательного программирования. Основной пример — логическое предусловие класса не выполняется, например, когда инвариант класса не может быть установлен. (Вспомните из главы 2, что инвариант класса — это особенность класса, которая всегда верна.)

Поскольку инвариант класса — это то, что определяет программист, ни компилятор, ни среда выполнения не могут применять его без посторонней помощи. Можно использовать конструктор класса для проверки различных условий, и если нельзя установить инвариант класса, можно вызвать исключение. Если сбой является результатом, скажем, передачи неверного параметра в конструктор, `logic_error` является подходящим типом исключения.

`logic_error` имеет несколько подклассов, о которых следует знать:

- `domain_error` сообщает об ошибках, связанных с допустимым диапазоном ввода, особенно для математических функций. Например, квадратный корень поддерживает только неотрицательные числа (в реальном случае). Если передается отрицательный аргумент, функция квадратного корня может выдать `domain_error`.
- Исключение `invalid_argument` сообщает, как правило, о неожиданных параметрах.
- Исключение `length_error` сообщает, что какое-либо действие нарушит ограничение максимального размера.
- Исключение `out_of_range` сообщает, что некоторое значение не находится в ожидаемом диапазоне. Каноническим примером является индексирование с проверкой границ в структуре данных.

Ошибки выполнения

Ошибки выполнения происходят из класса `runtime_error`. Эти исключения помогают сообщать об ошибках, которые выходят за рамки программы. Как и `logic_error`, `runtime_error` имеет несколько подклассов, которые могут оказаться полезными:

- `system_error` сообщает, что операционная система обнаружила некоторую ошибку. Такого рода исключения могут тысячи раз встретиться на вашем пути. Внутри заголовка `<system_error>` находится большое количество *кодов ошибок* и их *состояний*. Когда создается `system_error`, информация об ошибке упаковывается, чтобы можно было определить природу ошибки. Метод `.code()` возвращает `enumclass` типа `std::errc`, который имеет большое количество значений, таких как `bad_file_descriptor`, `timed_out` и `license_denied`,
- `overflow_error` и `underflow_error` сообщают об арифметическом переполнении и потере значимости соответственно.

Другие ошибки наследуются напрямую от `exception`. Распространенным является исключение `bad_alloc`, которое сообщает, что `new` не удалось выделить необходимую память для динамического хранения.

Ошибки языковой поддержки

Ошибки языковой поддержки не будут использоваться напрямую. Они существуют, чтобы указывать, что некоторые основные функции языка были неудачно использованы во время выполнения.

Обработка исключений

Правила обработки исключений основаны на наследовании классов. Когда выбрасывается исключение, блок `catch` обрабатывает его, если тип выброшенного исключения соответствует типу исключения обработчика или если тип выброшенного исключения *наследуется от* типа исключения обработчика.

Например, следующий обработчик перехватывает любое исключение, которое наследуется от `std::exception`, включая `std::logic_error`:

```
try {
  throw std::logic_error{ "It's not about who wrong "
                          "it's not about who right" };
} catch (std::exception& ex) {
  // Обрабатывает std::logic_error. Поскольку он наследуется от std::exception
}
```

Следующий специальный обработчик перехватывает *любое* исключение независимо от его типа:

```
try {
  throw 'z'; // Don't do this.
} catch (...) {
  // Обрабатывает любое исключение, даже 'z'
}
```

Специальные обработчики обычно используются в качестве механизма обеспечения безопасности для регистрации катастрофического сбоя программы при обнаружении исключения определенного типа.

Можно обрабатывать различные типы исключений, происходящих из одного и того же блока `try`, объединяя операторы `catch`, как показано здесь:

```
try {
  // Код, который может вызвать исключение
  --пропуск--
} catch (const std::logic_error& ex) {
  // Запись исключения и завершение работы программы; найдена программная ошибка!
  --пропуск--
} catch (const std::runtime_error& ex) {
  // Делаем все, что можно
  --пропуск--
} catch (const std::exception& ex) {
  // Обработка любого исключения, наследуемого от std:exception,
  // которое не является logic_error или runtime_error.
  --пропуск--
} catch (...) {
  // Паника; было сгенерировано непредвиденное исключение
  --пропуск--
}
```

Обычно такой код можно увидеть в точке входа в программу.

ПЕРЕБРАСЫВАНИЕ ИСКЛЮЧЕНИЯ

В блоке `catch` можно использовать ключевое слово `throw`, чтобы возобновить поиск подходящего обработчика исключений. Это называется *перебрасыванием исключения*. Есть несколько необычных, но важных случаев, когда вы, возможно, захотите дополнительно проверить исключение, прежде чем обработать его, как показано в листинге 4.11.

Листинг 4.11. Перебрасывание ошибки

```
try {
  // Код, который может вызвать system_error
  --пропуск--
} catch(const std::system_error& ex) {
  if(ex.code()!= std::errc::permission_denied){
    // Ошибка, не связанная с отказом в доступе
    throw; ❶
  }
  // Восстановление после ошибки
  --пропуск--
}
```

В этом примере код, который может выдать `system_error`, помещается в блок `try-catch`. Все системные ошибки обрабатываются, но, если это не ошибка EACCES (в доступе отказано), исключение перебрасывается ❶. У этого подхода есть некоторые потери производительности, и полученный код часто оказывается излишне запутанным.

Вместо повторной обработки можно определить новый тип исключения и создать отдельный обработчик перехвата для ошибки `EACCES`, как показано в листинге 4.12.

Листинг 4.12. Перехват конкретного исключения, но не перебрасывание

```
try {
  // Генерация исключения PermissionDenied
  --пропуск--
} catch(const PermissionDenied& ex) {
  // Восстановление после ошибки EACCES (отказано в доступе) ❶
  --пропуск--
}
```

Если генерируется `std::system_error`, обработчик `PermissionDenied` ❶ не поймает его. (Конечно, обработчик `std::system_error` все равно можно оставить, чтобы перехватывать такие исключения, если это необходимо.)

Пользовательские исключения

Программист может при необходимости определить свои собственные исключения; обычно эти *пользовательские исключения* наследуются от `std::exception`. Все классы из stdlib используют исключения, которые происходят от `std::exception`.

Это позволяет легко перехватывать все исключения, будь то из вашего кода или из stdlib, с помощью одного блока `catch`.

Ключевое слово noexcept

Ключевое слово `noexcept` — еще один термин, связанный с исключениями, который следует знать. Можно и нужно пометить любую функцию, которая в теории не может вызвать исключение, ключевым словом `noexcept`, как показано ниже:

```
bool is_odd(int x) noexcept {
  return 1 == (x % 2);
}
```

Функции с пометкой `noexcept` составляют жесткий контракт. При использовании функции, помеченной как `noexcept`, вы можете быть уверены, что функция не может вызвать исключение. В обмен на это вы должны быть предельно осторожны, когда помечаете собственную функцию как `noexcept`, так как компилятор не может это проверить. Если код выдает исключение внутри функции, помеченной как `noexcept`, это плохо. Среда выполнения C++ вызовет функцию `std::terminate`, которая по умолчанию завершит работу программы через `abort`. После такого программа не может быть восстановлена:

```
void hari_kari() noexcept {
  throw std::runtime_error{ "Goodbye, cruel world." };
}
```

Пометка функции ключевым словом `noexcept` позволяет оптимизировать код, полагаясь на то, что функция не может вызвать исключение. По сути, компилятор освобождается для использования семантики переноса, что может быть выполнено быстрее (подробнее об этом в разделе «Семантика перемещения», с. 184).

ПРИМЕЧАНИЕ

Ознакомьтесь с правилом 14 «Эффективного использования C++» Скотта Мейерса, чтобы подробно обсудить `noexcept`. Суть в том, что некоторые конструкторы переноса и операторы присваивания переноса могут выдавать исключение, например если им нужно выделить память, а система не работает. Если конструктор переноса или оператор присваивания переноса не указывает иное, компилятор должен предполагать, что перенос может вызвать исключение. Это отключает определенные оптимизации.

Исключения и стеки вызовов

Стек вызовов — это структура времени выполнения, в которой хранится информация об активных функциях. Когда часть кода (*вызывающая сторона*) вызывает функцию (*вызываемая сторона*), машина отслеживает, кто кого вызвал, помещая информацию в стек вызовов. Это позволяет программам иметь много вызовов функций,

вложенных друг в друга. Затем вызываемая функция может, в свою очередь, стать вызывающей, вызвав другую функцию.

Стеки

Стек — это гибкий контейнер данных, который может содержать динамическое количество элементов. Существуют две основные операции, которые поддерживаются всеми стеками: *вставка* элементов в верхнюю часть стека и *удаление* этих элементов. Эта структура данных организована по принципу «последним пришел — первым вышел», как показано на рис. 4.2.

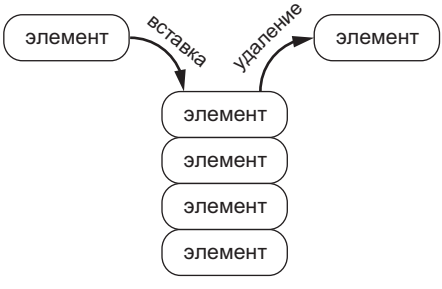


Рис. 4.2. Элементы, помещаемые в стек и извлекаемые из него

Как следует из названия, стек вызовов функционально похож на контейнер с одноименными данными. Каждый раз, когда вызывается функция, информация о вызове функции размещается в *кадре стека* и помещается в стек вызовов. Поскольку новый кадр стека помещается в стек для каждого вызова функции, вызываемый элемент может свободно вызывать другие функции, образуя произвольно глубокие цепочки вызовов. Всякий раз, когда функция возвращается, ее кадр стека выталкивается с вершины стека вызовов и управление выполнением возобновляется, как указано в предыдущем кадре стека.

Стеки вызовов и обработка исключений

Среда выполнения ищет ближайший обработчик исключений для выброшенного исключения. Если в текущем кадре стека есть соответствующий обработчик исключения, он его обработает. Если соответствующий обработчик не найден, среда выполнения раскручивает стек вызовов, пока не найдет подходящий обработчик. Любые объекты, срок жизни которых заканчивается, уничтожаются обычным способом.

Выбрасывание исключений из деструктора

Выбрасывание исключений из деструктора — это жонглирование бензопилами. Такое исключение обязательно должно быть поймано внутри деструктора.

Допустим, генерируется исключение и во время размотки стека другое исключение генерируется деструктором во время обычной очистки. Теперь у вас есть *два*

исключения в состоянии полета. Как среда выполнения C++ должна справляться с такой ситуацией?

У вас может быть свое мнение на этот счет, но среда выполнения вызовет функцию `terminate` (завершение). Рассмотрим листинг 4.13, который показывает, что может произойти при выбрасывании исключений из деструктора:

Листинг 4.13. Программа, где показана опасность создания исключения в деструкторе

```
#include <cstdio>
#include <stdexcept>

struct CyberdyneSeries800 {
  CyberdyneSeries800() {
    printf("I'm a friend of Sarah Connor."); ❶
  }
  ~CyberdyneSeries800() {
    throw std::runtime_error{ "I'll be back." }; ❷
  }
};

int main() {
  try {
    CyberdyneSeries800 t800; ❸
    thro std::runtime_error{ "Come with me if you want to live." }; ❹
  } catch(const std::exception& e) { ❺
    printf("Caught exception: %s\n", e.what()); ❻
  }
}
--------------------------------------------------------------------------------
I'm a friend of Sarah Connor. ❶
```

ПРИМЕЧАНИЕ

Листинг 4.13 вызывает `std::terminate`, поэтому в зависимости от операционной среды может быть показано всплывающее окно с уведомлением.

Во-первых, был объявлен класс `CyberdyneSeries800`, который имеет простой конструктор, который выводит сообщение ❶, и воинственный деструктор, который генерирует необработанное исключение ❷. В `main` определяется блок `try`, в котором инициализируется `CyberdyneSeries800` под именем `t800` ❸, и выбрасывается `runtime_error` ❹. В лучшем случае блок `catch` ❺ обработает это исключение, выведет его сообщение ❻ и все выйдет изящно. Поскольку `t800` — это автоматическая переменная в блоке `try`, она разрушается во время обычного процесса поиска обработчика для исключения, которое было выброшено ❹. А поскольку `t800` создает исключение в своем деструкторе ❷, программа вызывает `std::terminate` и внезапно завершается.

Как правило, обращайтесь с деструкторами так, как если бы они были `noexcept`.

Класс SimpleString

Используя расширенный пример, давайте рассмотрим, как конструкторы, деструкторы, члены и исключения объединяются. Класс `SimpleString` в листинге 4.14 позволяет добавлять строки в стиле C и выводить результат.

Листинг 4.14. Конструктор и деструктор класса SimpleString

```
#include <stdexcept>

struct SimpleString {
  SimpleString(size_t max_size) ❶
    : max_size{ max_size }, ❷
      length{} { ❸
    if (max_size == 0) {
      throw std::runtime_error{ "Max size must be at least 1." }; ❹
    }
    buffer = new char[max_size]; ❺
    buffer[0] = 0; ❻
  }

  ~SimpleString() {
    delete[] buffer; ❼
  }
--пропуск--
private:
  size_t max_size;
  char* buffer;
  size_t length;
};
```

Конструктор ❶ принимает один параметр `max_size`. Это максимальная длина строки, которая включает символ завершения строки. Инициализатор члена ❷ сохраняет эту длину в переменной-члене `max_size`. Это значение также используется в выражении `new` массива для выделения буфера для хранения данной строки ❺. Полученный указатель сохраняется в `buffer`. Длина инициализируется нулем ❸, и это гарантирует, что по крайней мере буфер будет достаточного размера для хранения нулевого байта ❹. Поскольку строка изначально пуста, первый байт буфера заполняется нулем ❻.

ПРИМЕЧАНИЕ

Поскольку `max_size` — это `size_t`, он не имеет знака и не может быть отрицательным, поэтому не нужно проверять это фиктивное условие.

Класс `SimpleString` владеет ресурсом — памятью, на которую указывает буфер, — которая должна быть освобождена при прекращении использования. Деструктор содержит одну строку ❼, которая освобождает `buffer`. Поскольку распределение и освобождение `buffer` связаны конструктором и деструктором `SimpleString`, память никогда не будет потеряна.

Этот шаблон называется «*получение ресурса есть инициализация*» (RAII), или «*получение конструктора — освобождение деструктора*» (CADRe).

ПРИМЕЧАНИЕ

Класс `SimpleString` все еще имеет неявно определенный конструктор копирования. Несмотря на то что память не может быть потеряна, при копировании класс потенциально освободится вдвое. Вы узнаете о конструкторах копирования в разделе«Семантике копирования», с. 176. Просто знайте, что листинг 4.14 — это обучающий инструмент, а не рабочий код.

Добавление и вывод

Класс `SimpleString` пока не очень полезен. В листинг 4.15 добавлена возможность выводить строку и добавлять набор символов в конец строки.

Листинг 4.15. Методы print и append_line для SimpleString

```
#include <cstdio>
#include <cstring>
#include <stdexcept>

struct SimpleString {
  --пропуск--
  void print(const char* tag) const { ❶
    printf("%s: %s", tag, buffer);
  }

  bool append_line(const char* x) { ❷
    const auto x_len = strlen❸(x);
    if (x_len + length + 2 > max_size) return false; ❹
    std::strncpy❺(buffer + length, x, max_size - length);
    length += x_len;
    buffer[length++] = '\n';
    buffer[length] = 0;
    return true;
  }
--пропуск--
};
```

Первый метод `print` ❶ выводит строку. Для удобства можно предоставить строку `tag`, чтобы можно было сопоставить вызов `print` с результатом. Этот метод является постоянным, потому что нет необходимости изменять состояние `SimpleString`.

Метод `append_line` ❷ принимает строку с нулем в конце и добавляет ее содержимое — плюс символ новой строки — в `buffer`. Он возвращает `true`, если был успешно добавлен, и `false`, если не было достаточно места. Во-первых, `append_line` должен определить длину `x`. Для этого используется функция `strlen` ❸ из заголовка

<cstring>, которая принимает строку с нулевым символом в конце и возвращает ее длину:

```
size_t strlen(const char* str);
```

strlen используется для вычисления длины x и инициализации x_len с результатом. Этот результат используется для вычисления того, приведет ли добавление x (символов новой строки) и нулевого байта к текущей строке к получению строки с длиной, превышающей max_size ❹. Если это так, append_line возвращает false.

Если для добавления x достаточно места, необходимо скопировать его байты в правильное место в buffer. Функция std::strncpy ❺ из заголовка <cstring> является одним из подходящих инструментов для этой работы. Она принимает три параметра: адрес назначения, адрес источника и количество символов для копирования:

```
char* std::strncpy(char* destination, const char* source, std::size_t num);
```

Функция strncpy будет копировать до num байтов из source в destination. После завершения она вернет значение destination (которое будет отброшено).

После добавления количества байтов x_len, скопированных в buffer, к length работа завершается добавлением символа новой строки \n и нулевого байта в конец buffer. Функция возвращает true, чтобы указать, что введенный x был успешно добавлен в виде строки в конец буфера.

ПРЕДУПРЕЖДЕНИЕ

Используйте strncpy очень осторожно. Слишком легко забыть символ конца строки в исходной строке или не выделить достаточно места в целевой строке. Обе ошибки приведут к неопределенному поведению. Мы рассмотрим более безопасную альтернативу во второй части книги.

Использование SimpleString

Листинг 4.16 показывает пример использования SimpleString, где добавляются несколько строк и промежуточные результаты выводятся в консоль.

Листинг 4.16. Методы SimpleString

```
#include <cstdio>
#include <cstring>
#include <exception>

struct SimpleString {
  --пропуск--
}
int main() {
  SimpleString string{ 115 }; ❶
  string.append_line("Starbuck, whaddya hear?");
```

```
    string.append_line("Nothin' but the rain."); ❷
    string.print("A"); ❸
    string.append_line("Grab your gun and bring the cat in.");
    string.append_line("Aye-aye sir, coming home."); ❹
    string.print("B"); ❺
    if (!string.append_line("Galactica!")) { ❻
      printf("String was not big enough to append another message."); ❼
    }
  }
```

Сначала создается `SimpleString` с `max_length=115` ❶. Метод `append_line` используется дважды ❷, чтобы добавить некоторые данные в строку, а затем вывести содержимое вместе с тегом `A` ❸. Затем добавляется больше текста ❹ и снова выводится содержимое, на этот раз с тегом `B` ❺. Когда `append_line` определяет, что `SimpleString` исчерпал свободное пространство ❻, возвращается `false` ❼. (Вы как пользователь `SimpleString` несете ответственность за проверку этого условия.)

Листинг 4.17 содержит выходные данные запуска этой программы.

Листинг 4.17. Результат выполнения программы в листинге 4.16

```
A: Starbuck, whaddya hear? ❶
Nothin' but the rain.
B: Starbuck, whaddya hear? ❷
Nothin' but the rain.
Grab your gun and bring the cat in.
Aye-aye sir, coming home.
String was not big enough to append another message. ❸
```

Как и ожидалось, строка содержит `Starbuck, whaddya hear?\nNothin' but the rain.\nвA` ❶. (Вспомните из главы 2, что `\n` — это специальный символ новой строки.) После добавления `Grab your gun and bring the cat in.` и `Aye-aye sir, coming home.` вы получите ожидаемый результат в `B` ❷.

Когда листинг 4.17 пытается добавить `Galactica!` в `string`, `append_line` возвращает `false`, поскольку в `buffer` недостаточно места. Это вызывает вывод сообщения `String was not big enough to append another message` ❸.

Составление SimpleString

Рассмотрим, что происходит при определении класса с членом `SimpleString`, как показано в листинге 4.18.

Как предполагает инициализатор члена ❶, `string` полностью построена, и ее инварианты класса назначаются после выполнения конструктора `SimpleStringOwner`. Здесь демонстрируется порядок членов объекта во время создания: *члены создаются перед вызовом конструктора окружающего объекта*. Смысл есть, а иначе как можно установить инварианты класса без знаний об инвариантах его членов?

Листинг 4.18. Реализация SimpleStringOwner

```
#include <stdexcept>

struct SimpleStringOwner {
  SimpleStringOwner(const char* x)
    : string{ 10 } { ❶
    if (!string.append_line(x)) {
      throw std::runtime_error{ "Not enough memory!" };
    }
    string.print("Constructed");
  }
  ~SimpleStringOwner() {
    string.print("About to destroy"); ❷
  }
private:
  SimpleString string;
};
```

Деструкторы работают в обратном порядке. Внутри `~SimpleStringOwner()` ❷ нужно хранить инварианты класса строки, чтобы можно было напечатать ее содержимое. *Все члены уничтожаются после вызова деструктора объекта.*

В листинге 4.19 используется `SimpleStringOwner`.

Листинг 4.19. Программа, содержащая SimpleStringOwner

```
--пропуск--
int main() {
  SimpleStringOwner x{ "x" };
  printf("x is alive\n");
}
-------------------------------------------------------------------------------
Constructed: x ❶
x is alive
About to destroy: x ❷
```

Как и ожидалось, член `string` в `x` ❶ создается надлежащим образом, потому что конструкторы членов объекта вызываются перед конструктором объекта, в результате чего появляется сообщение `Constructed: x.` Как автоматическая переменная `x` уничтожается непосредственно перед выходом из `main`, и вы получаете сообщение `About to destroy: x` ❷. Член `string` все еще доступен в этот момент, потому что деструкторы членов вызываются после деструктора вмещающего объекта.

Размотка стека вызовов

Листинг 4.20 демонстрирует, как обработка исключений и размотка стека работают вместе. Блок `try-catch` устанавливается в `main`, после чего выполняется серия вызовов функций. Один из этих вызовов вызывает исключение.

Листинг 4.20. Программа, где используется SimpleStringOwner и размотка стека вызовов

```
--пропуск--
void fn_c() {
  SimpleStringOwner c{ "ccccccccc" }; ❶
}

void fn_b() {
  SimpleStringOwner b{ "b" };
  fn_c(); ❷
}

int main() {
  try { ❸
    SimpleStringOwner a{ "a" };
    fn_b(); ❹
    SimpleStringOwner d{ "d" }; ❺
  } catch(const std::exception& e) { ❻
    printf("Exception: %s\n", e.what());
  }
}
```

В листинге 4.21 показаны результаты запуска программы из листинга 4.20.

Листинг 4.21. Результат запуска программы из листинга 4.20

```
Constructed: a
Constructed: b
About to destroy: b
About to destroy: a
Exception: Not enough memory!
```

Вы установили блок `try-catch` ❸. Первый экземпляр `SimpleStringOwner`, `a`, создается без инцидентов, и в консоль выводится сообщение `Constructed: a`. Далее вызывается `fn_b` ❹. Обратите внимание, что вы все еще находитесь в блоке `try-catch`, поэтому любое выброшенное исключение будет обработано. Внутри `fn_b` другой экземпляр `SimpleStringOwner`, `b`, успешно создается, и `Constructed: b` выводится на консоль. Затем происходит вызов еще одной функции, `fn_c` ❷.

Давайте на минуту остановимся, чтобы разобраться, как выглядит стек вызовов, какие объекты живы и как выглядит ситуация обработки исключений. Сейчас у нас есть два живых и действительных объекта `SimpleStringOwner`: `a` и `b`. Стек вызовов выглядит как `main() → fn_ () → fn_c()`, и в `main` настроен обработчик исключений для обработки любых исключений. Эта ситуация показана на рис. 4.3.

В ❶ возникает небольшая проблема. Напомним, что `SimpleStringOwner` имеет член `SimpleString`, который всегда инициализируется с `max_size` 10. При попытке создания `c` конструктор `SimpleStringOwner` выдает исключение, потому что вы пытались добавить «`ccccccccc`», который имеет длину 10, что выходит за рамки, потому что нужно еще добавить символы новой строки и завершения строки.

Теперь в полете находится одно исключение. Стек будет раскручиваться до тех пор, пока не будет найден соответствующий обработчик, и все объекты, выпадающие из области видимости в результате этого раскручивания, будут уничтожены. Обработчик доходит до стека ❻, поэтому `fn_c` и `fn_b` разматываются. Поскольку `SimpleStringOwner b` — это автоматическая переменная в `fn_b`, она разрушается и в консоль выводится сообщение `About to destroy: b`. После `fn_b` автоматические переменные внутри `try {}` уничтожаются. Это включает в себя `SimpleStringOwner a`, поэтому в консоль выводится `About to destroy: a.`

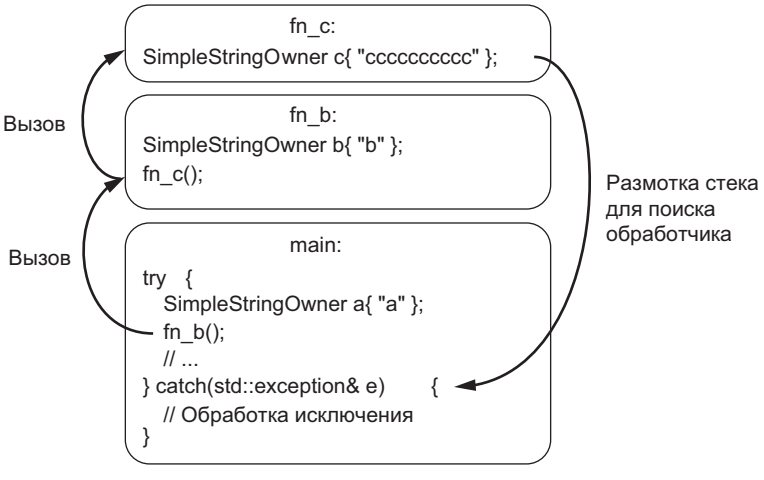


Рис. 4.3. Стек вызовов на момент вызова функцией fn_c конструктора

Как только исключение происходит в блоке `try{}`, дальнейшие операторы не выполняются. В результате `d` никогда не инициализируется ❺ и конструктор `d` не вызывается и не выводится в консоль. После размотки стека вызовов выполнение сразу переходит к блоку `catch`. В итоге в консоль выводится сообщение `Exception: Not enough memory!` ❻.

Исключения и производительность

Обработка исключений обязательна в программах; ошибки неизбежны. При правильном использовании исключений ошибок не возникает, код работает быстрее, чем код, проверенный вручную. Если ошибка все-таки есть, обработка исключений может иногда выполняться медленнее, но у этого есть огромные преимущества в надежности и удобстве обслуживания по сравнению с альтернативными вариантами. Курт Гантерот, автор «*Оптимизации программ на C++*»[1], хорошо пишет об этом: «Использование обработки исключений приводит к программам, которые рабо-

[1] *Гантерот К.* Оптимизация программ на C++. Проверенные методы повышения производительности. — Вильямс, 2017. — 400 с.

тают быстрее при нормальном выполнении и ведут себя лучше в случае неудачи». Когда программа на C++ выполняется нормально (без исключений), при проверке исключений не возникает никаких издержек во время выполнения. Вы платите только за исключение.

Надеюсь, вы убедились в центральной роли, которую играют исключения в идиоматических программах на C++. К сожалению, иногда нет возможности использовать исключения. Одним из примеров является встроенная разработка, где требуются гарантии в реальном времени. Инструменты просто не существуют (пока) для этих настроек. Если повезет, это скоро изменится, но сейчас приходится обходиться без исключений в большинстве встроенных контекстов. Другой пример — некоторый устаревший код. Исключения изящны из-за того, как они вписываются в объекты RAII. Когда деструкторы отвечают за очистку ресурсов, раскрутка стека является прямым и эффективным способом защиты от утечек памяти. В устаревшем коде можно найти ручное управление ресурсами и обработку ошибок вместо объектов RAII. Это делает использование исключений очень опасным, поскольку размотка стека безопасна только для объектов RAII. Без них можно с легкостью допустить утечку ресурсов.

Альтернативы для исключений

В ситуациях, когда исключения недоступны, не все потеряно. Хотя необходимо отслеживать ошибки вручную, существуют некоторые полезные функции C++, которые можно использовать, чтобы немного это исправить. Во-первых, можно вручную применить инварианты класса, предоставив некоторый метод, который сообщает, могут ли инварианты класса быть установлены, как показано здесь:

```
struct HumptyDumpty {
   HumptyDumpty();
   bool is_together_again();
  --пропуск--
};
```

В идиоматическом C++ вы бы просто сгенерировали исключение в конструкторе, но здесь следует помнить о проверке и обработке ситуации как условии ошибки в вызывающем коде:

```
bool send_kings_horses_and_men() {
  HumptyDumpty hd{};
  if (hd.is_together_again()) return false;
  // Использование инвариантов класса hd гарантировано.
  // HumptyDumpty с треском проваливается.
  --пропуск--
  return true;
}
```

Вторая, дополнительная, стратегия копирования — возвращать несколько значений с помощью *объявления структурированной привязки*, языковой функции, которая

позволяет возвращать несколько значений из вызова функции. Эту функцию можно использовать для возврата флагов успеха вместе с обычным возвращаемым значением, как показано в листинге 4.22.

Листинг 4.22. Фрагмент кода с объявлением структурированной привязки

```
struct Result { ❶
  HumptyDumpty hd;
  bool success;
};

Result make_humpty() { ❷
  HumptyDumpty hd{};
  bool is_valid;
  // Проверка правильности hd и установка соответствующего значения is_valid
  return { hd, is_valid };
}

bool send_kings_horses_and_men() {
  auto [hd, success] = make_humpty(); ❸
  if(!success) return false;
  // Установка инвариантов класса
  --пропуск--
  return true;
}
```

Сначала объявляется POD, который содержит `HumptyDumpty` и флаг `success` ❶. Затем определяется функция `make_humpty` ❷, которая создает и проверяет `HumptyDumpty`. Такие методы называются фабричными, поскольку их целью является инициализация объектов. Функция `make_humpty` оборачивает его и флаг `success` в `Result` при возврате. Синтаксис в точке вызова ❸ показывает, как можно распаковать `Result`, получив несколько переменных с определением типа при помощи `auto`.

ПРИМЕЧАНИЕ

Более подробное описание структурированных привязок приведено в подразделе «Структурированные привязки», с. 289.

Семантика копирования

Семантика копирования — это «смысл копирования». На практике программисты используют этот термин для обозначения правил создания копий объектов: после того как `x` *скопирован в* `y`, они *эквивалентны* и *независимы*. То есть `x == y` имеет значение `true` после копирования (эквивалентность), а изменение `x` не вызывает изменение `y` (независимость). Копирование чрезвычайно распространено, особенно при передаче объектов в функции по значению, как показано в листинге 4.23.

Листинг 4.23. Программа, где пока что передача по значению генерирует копию

```
#include <cstdio>

int add_one_to(int x) {
  x++; ❶
  return x;
}

int main() {
  auto original = 1;
  auto result = add_one_to(original); ❷
  printf("Original: %d; Result: %d", original, result);
}
--------------------------------------------------------------------------------
Original: 1; Result: 2
```

Здесь функция `add_one_to` принимает параметр `x` по значению. Затем она изменяет значение `x` ❶. Эта модификация изолирована от вызывающего ❷; `original` не затронут, потому что `add_one_to` получает копию.

Для пользовательских типов POD история аналогична. Передача по значению приводит к тому, что каждое значение элемента копируется в параметр (посимвольная копия), как показано в листинге 4.24.

Листинг 4.24. Функция make_transpose генерирует копию типа POD Point

```
struct Point {
  int x, y;
};

Point make_transpose(Point p) {
  int tmp = p.x;
  p.x = p.y;
  p.y = tmp;
  return p;
}
```

При вызове функция `make_transpose` получает копию `Point` в `p`, и оригинал не изменяется.

Для фундаментальных и POD-типов история проста. Копирование этих типов выполняется для каждого элемента, а это означает, что каждый элемент копируется в соответствующий пункт назначения. Это фактически побитовая копия с одного адреса памяти на другой.

Полнофункциональные классы требуют больше размышлений. Семантика копирования по умолчанию для полнофункциональных классов также является копией для каждого элемента, и это может быть чрезвычайно опасно. Снова рассмотрим класс `SimpleString`. Произошла бы катастрофа, если бы вы позволили пользователю сделать поочередную копию живого класса `SimpleString`. Два класса `SimpleString`

будут указывать на один и тот же `buffer`. Поскольку обе копии добавляются в один и тот же `buffer`, они сотрут друг друга. На рис. 4.4 приведена эта ситуация.

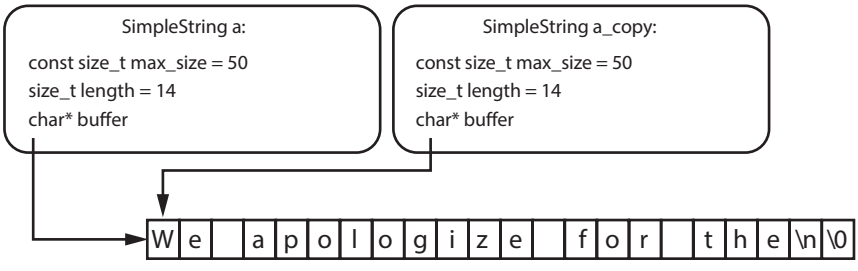


Рис. 4.4. Изображение семантики копирования по умолчанию в классе SimpleString

Такой результат плох, но еще хуже, если классы `SimpleString` начинают уничтожаться. Когда один из классов `SimpleString` будет уничтожен, буфер освободится. Когда оставшийся класс `SimpleString` пытается записать свой буфер — бабах! — возникнет неопределенное поведение. В какой-то момент этот оставшийся класс `SimpleString` будет разрушен и буфер снова освободится, что приведет к так называемому двойному освобождению.

ПРИМЕЧАНИЕ

Как и его злобный кузен —использование после освобождения, двойное освобождение может привести к неуловимым и трудно диагностируемым ошибкам, которые проявляются очень редко. Двойное освобождение происходит, если объект дважды освобождается. Напомним, что после освобождения объекта срок его хранения заканчивается. Память теперь находится в неопределенном состоянии, и если объект, который уже был разрушен, уничтожается, это приводит к неопределенному поведению. В некоторых ситуациях это может привести к серьезным уязвимостям безопасности.

Если взять под контроль семантику копирования, такой ситуации можно избежать. Укажите конструкторы копирования и операторы копирования, как описано в следующих разделах.

Конструкторы копирования

Существуют два способа скопировать объект. Одним из них является использование *конструкции копирования*, которая создает копию и присваивает ее совершенно новому объекту. Конструктор копирования выглядит, как другие конструкторы:

```
struct SimpleString {
  --пропуск--
  SimpleString(const SimpleString& other);
};
```

Обратите внимание, что `other` имеет пометку `const`. Вы копируете оригинальную `SimpleString`, и не существует причин для ее изменения. Конструктор копирования используется точно так же, как и другие конструкторы с унифицированным синтаксисом инициализации фигурных инициализаторов:

```
SimpleString a;
SimpleString a_copy{ a };
```

Вторая строка вызывает конструктор копирования `SimpleString` с `a` для получения `a_copy`.

Давайте реализуем конструктор копирования `SimpleString`. Необходима так называемая глубокая копия, когда данные, на которые указывает исходный буфер, копируются в новый буфер, как показано на рис. 4.5.

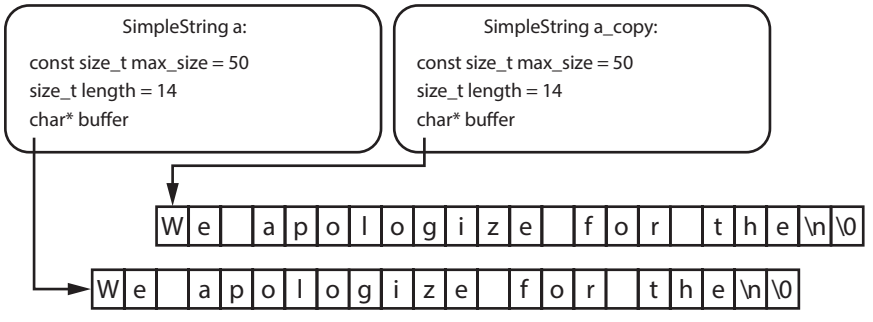


Рис. 4.5. Глубокое копирование класса SimpleString

Вместо того чтобы копировать указатель `buffer`, создадим новое выделение памяти в свободном хранилище, а затем скопируем все данные, на которые указывает исходный буфер. В результате получатся два независимых класса `SimpleString`. Листинг 4.25 реализует конструктор копирования `SimpleString`.

Листинг 4.25. Конструктор копирования класса SimpleString

```
SimpleString(const SimpleString& other)
  : max_size{ other.max_size }, ❶
    buffer{ new char[other.max_size] }, ❷
    length{ other.length } { ❸
    std::strncpy(buffer, other.buffer, max_size); ❹
}
```

Инициализаторы членов используются для `max_size` ❶, `buffer` ❷ и `length` ❸, которые передаются в соответствующие поля `other`. Можно использовать `new` для массивов ❷, чтобы инициализировать `buffer`, потому что вы знаете, что `other.max_size` больше 0. Тело конструктора копирования содержит единственный оператор ❹, копирующий содержимое, на которое указывает `other.buffer`, в массив, на который указывает `buffer`.

Листинг 4.26 использует этот конструктор копирования, инициализируя `SimpleString` существующим `SimpleString`.

Листинг 4.26. Программа, использующая конструктор копирования класса SimpleString

```
--пропуск--
int main() {
  SimpleString a{ 50 };
  a.append_line("We apologize for the");
  SimpleString a_copy{ a }; ❶
  a.append_line("inconvenience."); ❷
  a_copy.append_line("incontinence."); ❸
  a.print("a");
  a_copy.print("a_copy");
}
-------------------------------------------------------------------------------
a: We apologize for the
inconvenience.
a_copy: We apologize for the
incontinence.
```

В программе `SimpleStringa_copy` ❶ является копией, созданной из `a`. Она эквивалентна и независима от оригинала. Можно добавлять различные сообщения в конец `a` ❷ и `a_copy` ❸, и изменения будут изолированы.

Конструктор копирования вызывается при передаче `SimpleString` в функцию по значению, как показано в листинге 4.27.

Листинг 4.27. Программа, в которой конструкторы копирования вызываются при передаче объекта по значению

```
--пропуск--
void foo(SimpleString x) {
  x.append_line("This change is lost.");
}

int main() {
  SimpleString a { 20 };
  foo(a); // Вызов конструктора копирования
  a.print("Still empty");
}
-------------------------------------------------------------------------------
Still empty:
```

ПРИМЕЧАНИЕ

Не следует использовать передачу по значению, чтобы избежать непредвиденных изменений. Используйте ссылку `const`.

Влияние копирования на производительность может быть значительным, особенно в ситуации, когда задействовано выделение свободной памяти и копии буфера.

Допустим, есть класс, который управляет жизненным циклом гигабайта данных. Каждый раз при копировании объекта нужно выделить и скопировать гигабайт данных. Это может занять много времени, поэтому важно убедиться, что без копии не обойтись. Если будет достаточно передачи постоянной ссылки, настоятельно рекомендуем этим воспользоваться.

Присваивание копии

Другой способ сделать копию в C++ — использовать *оператор присваивания копии*. Можно создать копию объекта и назначить ее другому существующему объекту, как показано в листинге 4.28.

Листинг 4.28. Использование оператора присваивания копии по умолчанию для создания копии объекта и присваивания ее другому существующему объекту

```
--пропуск--
void dont_do_this() {
  SimpleString a{ 50 };
  a.append_line("We apologize for the");
  SimpleString b{ 50 };
  b.append_line("Last message");
  b = a; ❶
}
```

ПРИМЕЧАНИЕ

Код из листинга 4.28 вызывает неопределенное поведение, поскольку в нем отсутствует пользовательский оператор копирования присваивания.

Строка в ❶ *копирует присваивание* `a` в `b`. Основное различие между присваиванием копии и созданием копии состоит в том, что при присваивании копия `b` может уже иметь значение. Нужно очистить ресурсы `b` перед копированием `a`.

ПРЕДУПРЕЖДЕНИЕ

По умолчанию оператор присваивания копии для простых типов просто копирует элементы из исходного объекта в целевой. В случае с `SimpleString` это очень опасно по двум причинам. Во-первых, буфер исходного класса `SimpleString` перезаписывается без освобождения динамически размещаемого массива `char`. Во-вторых, теперь два класса `SimpleString` имеют один и тот же буфер, что может привести к висячим указателям и двойным освобождениям. Нужно реализовать оператор присваивания копии, который выполняет чистую передачу.

Оператор присваивания копии использует синтаксис `operator=`, как показано в листинге 4.29.

Оператор присваивания копии возвращает ссылку на результат, который всегда имеет значение `*this` ❷. Также обычно рекомендуется проверять, ссылается ли `other` на `this` ❶.

Листинг 4.29. Пользовательский оператор присваивания копии для SimpleString

```
struct SimpleString {
  --пропуск--
  SimpleString& operator=(const SimpleString& other) {
    if (this == &other) return *this; ❶
    --пропуск--
    return *this; ❷
  }
}
```

Можно реализовать присваивание копии для `SimpleString`, следуя этим рекомендациям: освободите `buffer` от `this`, а затем скопируйте `other`, как вы это делали при использовании конструктора копирования (листинг 4.30).

Листинг 4.30. Оператор присваивания копии для SimpleString

```
SimpleString& operator=(const SimpleString& other) {
  if (this == &other) return *this;
  const auto new_buffer = new char[other.max_size]; ❶
  delete[] buffer; ❷
  buffer = new_buffer; ❸
  length = other.length; ❹
  max_size = other.max_size; ❺
  std::strncpy(buffer, other.buffer, max_size); ❻
  return *this;
}
```

Оператор присваивания копии начинается с выделения `new_buffer` с соответствующим размером ❶. Далее `buffer` очищается ❷. Остальное по существу идентично конструктору копирования в листинге 4.25. `buffer` ❸, `length` ❹ и `max_size` ❺ копируются, а затем копируется содержимое из `other.buffer` в собственный `buffer` ❻.

В листинге 4.31 показано, как работает присваивание копии `SimpleString` (как реализовано в листинге 4.30).

Листинг 4.31. Программа, где происходит присваивание копии с помощью класса SimpleString

```
--пропуск--
int main() {
  SimpleString a{ 50 };
  a.append_line("We apologize for the"); ❶
  SimpleString b{ 50 };
  b.append_line("Last message"); ❷
  a.print("a"); ❸
  b.print("b"); ❹
  b = a; ❺
  a.print("a"); ❻
  b.print("b"); ❼
}
----------------------------------------------------------------------------
```

```
a: We apologize for the ❸
b: Last message ❹
a: We apologize for the ❻
b: We apologize for the ❼
```

Сначала объявляются два класса `SimpleString` с разными сообщениями: строка `a` содержит `We apologize for the` ❶, а `b` содержит `Last message` ❷. Эти строки выводятся в консоль, чтобы убедиться, что они содержат указанный текст ❸❹. Далее присваивается копия `b`, равная `a` ❺. Теперь `a` и `b` содержат копии одного и того же сообщения, `We apologize for the` ❻❼. Но — и это важно — это сообщение находится в двух разных местах памяти.

Копирование по умолчанию

Часто компилятор генерирует реализации по умолчанию для конструктора копирования и присваивания копии. Реализация по умолчанию — вызывать конструктор копирования или присваивание копии для каждого из членов класса. Каждый раз, когда класс управляет ресурсом, нужно быть чрезвычайно осторожным с семантикой копирования по умолчанию; скорее всего, класс ошибается (как было в примере с `SimpleString`). Согласно рекомендациям, нужно явно объявить, что присваивание копии и конструктор копирования по умолчанию приемлемы для таких классов с использованием ключевого слова `default`. Например, класс `Replicant` имеет семантику копирования по умолчанию, как показано здесь:

```
struct Replicant {
  Replicant(const Replicant&) = default;
  Replicant& operator=(const Replicant&) = default;
  --пропуск--
};
```

Некоторые классы просто не могут или не должны копироваться — например, если класс управляет файлом или если он представляет собой блокировку взаимного исключения для параллельного программирования. Можно запретить компилятору генерировать конструктор копирования и оператор присваивания копии, используя ключевое слово `delete`. Например, класс `Highlander` нельзя скопировать:

```
struct Highlander {
  Highlander(const Highlander&) = delete;
  Highlander& operator=(const Highlander&) = delete;
  --пропуск--
};
```

Любая попытка скопировать `Highlander` приведет к ошибке компилятора:

```
--пропуск--
int main() {
  Highlander a;
  Highlander b{ a }; // Бах! Допустим только один класс.
}
```

Настоятельно рекомендую явно определить оператор присваивания копии и конструктор копирования для *любого* класса, которому принадлежит ресурс (например, принтер, сетевое соединение или файл). Если пользовательское поведение не требуется, используйте либо `default`, либо `delete`. Это избавит вас от множества неприятных и трудных для отладки ошибок.

Руководство по копированию

При реализации поведения копирования стоит подумать о следующих критериях:

- **Корректность.** Нужно убедиться, что инварианты классов поддерживаются. Класс `SimpleString` продемонстрировал, что конструктор копирования по умолчанию может нарушать инварианты.
- **Независимость.** После присваивания копии или конструктора копирования исходный объект и копия не должны изменять состояние друг друга во время модификации. Если бы вы просто скопировали `buffer` из одной `SimpleString` в другую, запись в один `buffer` могла бы перезаписать данные из другого.
- **Эквивалентность.** Оригинал и копия должны быть *одинаковыми*. Семантика одинаковости зависит от контекста. Но, как правило, операция, примененная к оригиналу, должна давать тот же результат при применении к копии.

Семантика перемещения

Копирование может занимать много времени во время выполнения, когда задействован большой объем данных. Часто необходимо просто *передать право собственности* на ресурсы с одного объекта на другой. Можно сделать копию и уничтожить оригинал, но это зачастую неэффективно. Вместо этого можно *перенести* права на владение.

Семантика перемещения — это следствие для семантики копирования, и она требует, чтобы после того, как объект `y` был *перемещен в* объект `x`, `x` был эквивалентен предыдущему значению `y`. После переноса `y` находится в особом состоянии, которое называется *состоянием перемещения*. Можно выполнить только две операции над перемещенными объектами: (пере)назначить или уничтожить их. Обратите внимание, что перемещение объекта `y` в объект `x` — это не просто переименование: это отдельные объекты с отдельным хранилищем и потенциально отдельным временем жизни.

Подобно тому как задается поведение копирования, нужно указывать, как объекты перемещаются с помощью *конструкторов переноса* и *операторов присваивания перемещения*.

Копирование может быть расточительным

Предположим, нужно переместить `SimpleString` в `SimpleStringOwner` следующим образом:

```
--пропуск--
void own_a_string() {
  SimpleString a{ 50 };
  a.append_line("We apologize for the");
  a.append_line("inconvenience.");
  SimpleStringOwner b{ a };
  --пропуск--
}
```

Можно добавить конструктор для `SimpleStringOwner`, а затем скопировать-сконструировать его член `SimpleString`, как показано в листинге 4.32.

Листинг 4.32. Наивный подход к инициализации элемента, содержащий расточительное копирование

```
struct SimpleStringOwner {
  SimpleStringOwner(const SimpleString& my_string) : string{ my_string }❶ { }
  --пропуск--
private:
  SimpleString string; ❷
};
```

У этого подхода есть скрытые издержки. У вас есть конструкция копирования ❶, но вызывающая сторона никогда не использует объект, на который указывает указатель, после создания `string` ❷. На рис. 4.6 приведена такая проблема.

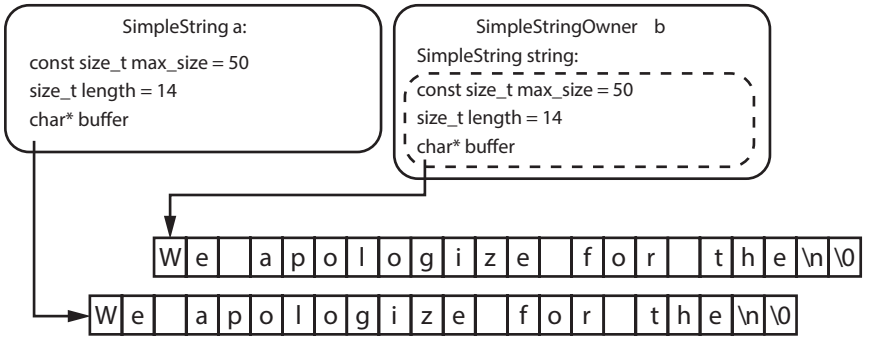


Рис. 4.6. Использование конструктора копирования для строки расточительно

Вы должны переместить начинку `SimpleString a` в строковое поле `SimpleStringOwner`. На рис. 4.7 показано, чего необходимо достичь: `SimpleStringOwner b` захватывает `buffer` и переводит `SimpleString a` в состояние уничтожения.

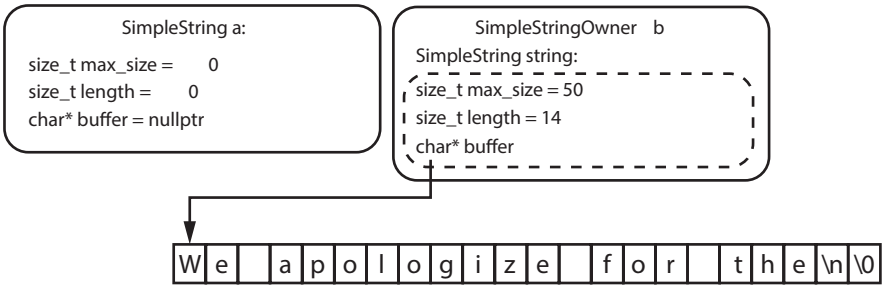


Рис. 4.7. Замена buffer из a в b

После перемещения `a` `SimpleString` из `b` эквивалентна прежнему состоянию `a`, в то время как экземпляр `a` становится уничтожаемым.

Перемещение может быть опасным. Если случайно использовать перенесенную переменную `a`, произойдет катастрофа. Инварианты класса `SimpleString` не удовлетворяются после переноса `a`.

К счастью, компилятор имеет встроенные средства защиты: `l`-значения и `r`-значения.

Категории значений

Каждое выражение имеет две важные характеристики: его *тип* и *категорию значения*. Категория значений описывает, какие виды операций допустимы для выражения. Благодаря эволюционному характеру C++ категории значений являются сложными: выражение может быть «обобщенным `l`-значением» (*gl-значением*), «простым `r`-значением» (*pr-значением*), «истекающим значением» (*x-значением*), «`l`-значением» (*gl-значением, которое не является x-значением*) или «`r`-значением» (*pr-значением или x-значением*). К счастью для новичка, не нужно много знать о большинстве этих категорий значений.

Мы рассмотрим очень упрощенное представление о категориях. На данный момент достаточного общего понимания `l`-значений и `r`-значений. `l`-значение — это любое значение, которое имеет имя, а `r`-значение — это все, что не является `l`-значением.

Рассмотрим следующие инициализации:

```
SimpleString a{ 50 };
SimpleStringOwner b{ a };                  // a – l-значение
SimpleStringOwner c{ SimpleString{ 50 } }; // SimpleString{ 50 } – r-значение
```

Этимология этих терминов — это *правое* (right) *значение* и *левое* (left) *значение*, относящиеся к тому, где они появляются по отношению к знаку равенства в конструкции. В утверждении `int x = 50;` `x` находится слева от знака равенства (`l`-значение), а `50` — справа от знака равенства (`r`-значение). Эти термины не совсем точны, потому что `l`-значение может находиться справа от знака равенства (например, при присваивании копии).

ПРИМЕЧАНИЕ

Стандарт ISO C++ детализирует категории значений в [basic] и [expr].

Ссылки на l-значения и r-значения

Можно сообщить компилятору, что функция принимает `l`-значения или `r`-значения, используя *ссылки на l-значения* и *r-значения*. До этого момента в этой книге каждый ссылочный параметр был ссылкой на `l`-значение и обозначался одним знаком `&`. Также можно получить параметр по ссылке `r`-значения, используя `&&`.

К счастью, компилятор отлично определяет, является ли объект `l`-значением или `r`-значением. Фактически можно определить несколько функций с одним и тем же именем, но с разными параметрами, и компилятор автоматически вызовет правильную версию в зависимости от того, какие параметры были предоставлены при вызове функции.

Листинг 4.33 содержит две функции с именем `ref_type`, чтобы определить, передал ли вызывающий объект ссылку на `l`-значение или `r`-значение.

Листинг 4.33. Программа, содержащая перегруженную функцию со ссылками на l-значение и r-значение

```
#include <cstdio>

void ref_type(int &x) { ❶
  printf("lvalue reference %d\n", x);
}

void ref_type(int &&x) { ❷
  printf("rvalue reference %d\n", x);
}

int main() {
  auto x = 1;
  ref_type(x); ❸
  ref_type(2); ❹
  ref_type(x + 2); ❺
}
--------------------------------------------------------------------------------
lvalue reference 1 ❸
rvalue reference 2 ❹
rvalue reference 3 ❺
```

Версия `int &x` ❶ принимает ссылку на `l`-значение, а версия `int &&x` ❷ — ссылку на `r`-значение. `ref_type` вызывается три раза. Сначала вызывается эталонная версия `l`-значения, потому что `x` является `l`-значением (у него есть имя) ❸. Во-вторых, вызывается эталонная версия `r`-значения, потому что 2 — это целочисленный литерал без имени ❹. В-третьих, результат добавления 2 к `x` не привязан к имени, поэтому это `r`-значение ❺.

ПРИМЕЧАНИЕ

Определение нескольких функций с одним и тем же именем, но с разными параметрами называется перегрузкой функций, которую мы подробно изучим в главе 9.

Функция std::move

Можно привести ссылку на `l`-значение к ссылке на `r`-значение, используя функцию `std::move` из заголовка `<utility>`. Листинг 4.34 обновляет листинг 4.33 для использования функция `std::move.`

Листинг 4.34. Рефакторинг листинга 4.33 с использованием std::move для приведения x к r-значению

```
#include <utility>
--пропуск--
int main() {
  auto x = 1;
  ref_type(std::move(x)); ❶
  ref_type(2);
  ref_type(x + 2);
}
-------------------------------------------------------------------------------
rvalue reference 1 ❶
rvalue reference 2
rvalue reference 3
```

Как и ожидалось, `std::move` изменяет `l`-значение `x` на `r`-значение ❶. Перегрузка `ref_type` с `l`-значением никогда не вызывается.

ПРИМЕЧАНИЕ

Комитет C++, вероятно, должен был бы назвать `std::move` как `std::rvalue`, но это имя нельзя было использовать. Функция `std:move` на самом деле ничего не перемещает — она выполняет приведение.

Будьте очень осторожны при использовании `std::move`, потому что вы отказываетесь от мер, не позволяющих взаимодействовать с перемещенным объектом. Можно выполнить два действия над перемещенным объектом: уничтожить его или переназначить.

Как семантика `l`-значений и `r`-значений включает семантику переноса, теперь должно быть ясно. Если под рукой находится `l`-значение, перенос отменяется. Если используется `r`-значение, перенос разрешен.

Конструктор переноса

Конструкторы переноса выглядят как конструкторы копирования, за исключением того, что они принимают ссылки на `r`-значения вместо ссылок на `l`-значения.

Рассмотрим конструктор перемещения `SimpleString` в листинге 4.35.

Листинг 4.35. Конструктор перемещения для SimpleString

```
SimpleString(SimpleString&& other) noexcept
  : max_size{ other.max_size }, ❶
  buffer(other.buffer),
  length(other.length) {
  other.length = 0; ❷
  other.buffer = nullptr;
  other.max_size = 0;
}
```

Поскольку `other` — это ссылка на `r`-значение, разрешено ее уничтожить. В случае с `SimpleString` это легко: просто скопируйте все поля `other` в `this` ❶, а затем обнулите поля `other` ❷. Последний шаг важен: он переводит `other` в перемещенное состояние. (Подумайте, что произойдет после уничтожения `other`, если не очистить его члены.)

Выполнение этого конструктора переноса намного дешевле, чем выполнение конструктора копирования.

Конструктор переноса разработан так, чтобы не создавать исключение, поэтому он помечается как `noexcept`. Предпочтительно использовать `noexcept` конструкторы переноса; часто компилятор не может использовать конструкторы переноса, генерирующие исключения, и вместо этого будет использовать конструкторы копирования. Компиляторы предпочитают медленный, но правильный код вместо быстрого и неправильного.

Присваивание перемещения

Также можно создать аналог перемещения для присваивания копии с помощью `operator=`. Оператор присваивания переноса принимает ссылку на `r`-значение, а не `const`-ссылку на `l`-значение, и она обычно помечается как исключение. В листинге 4.36 реализован такой оператор присваивания переноса для `SimpleString`.

Листинг 4.36. Оператор присваивания перемещения для SimpleString

```
SimpleString& operator=(SimpleString&& other) noexcept { ❶
  if (this == &other) return *this; ❷
  delete[] buffer; ❸
  buffer = other.buffer; ❹
  length = other.length;
  max_size = other.max_size;
  other.buffer = nullptr; ❺
  other.length = 0;
  other.max_size = 0;
  return *this;
}
```

Оператор присваивания перемещения объявляется с использованием синтаксиса ссылки на `r`-значение и спецификатора `noexcept`, как и в конструкторе перено-

са ❶. Проверка ссылки на самого себя ❷ обрабатывает присваивание переноса `SimpleString` для его самого. `buffer` ❸ очищается перед тем, как присваивать поля `this` полям `other` ❹, и поля `other` обнуляются ❺. Помимо проверки ссылки на самого себя ❷ и очистки ❸, оператор присваивания переноса и конструктор переноса функционально идентичны.

Теперь, когда `SimpleString` является переносимым, можно завершить конструктор `SimpleString` для `SimpleStringOwner`:

```
SimpleStringOwner(SimpleString&& x) : string{ std::move(x)❶ } { }
```

`x` является `l`-значением, поэтому нужно вызвать `std::move` `x` в конструкторе переноса `string` ❶. `std::move` может показаться труднообъяснимым, потому что `x` является ссылкой на `r`-значение. Напомним, что `l`-значения/`r`-значения и ссылки на `l`-значения/`r`-значения являются различными дескрипторами.

Подумайте, что было бы, если `std::move` не был нужен? Что, если бы вы переместились из `x`, а затем использовали его внутри конструктора? Это может привести к ошибкам, которые трудно диагностировать. Помните, что нельзя использовать перемещенные объекты никоим образом, кроме как для переназначения или уничтожения. Делать что-либо еще — неопределенное поведение. В листинге 4.37 показано присваивание переноса `SimpleString`.

Листинг 4.37. Программа, где происходит присваивание перемещения в классе SimpleString

```
--пропуск--
int main() {
  SimpleString a{ 50 };
  a.append_line("We apologize for the"); ❶
  SimpleString b{ 50 };
  b.append_line("Last message"); ❷
  a.print("a"); ❸
  b.print("b"); ❹
  b = std::move(a); ❺
  // a находится в "перемещенном" состоянии
  b.print("b"); ❻
}
-----------------------------------------------------------------------------
a: We apologize for the ❸
b: Last message ❹
b: We apologize for the ❻
```

Как и в листинге 4.31, все начинается с объявления двух классов `SimpleString` с разными сообщениями: строка `a` содержит `We apologize for the` ❶, а `b` содержит `Last message` ❷. Выведите эти строки, чтобы убедиться, что они содержат указанное ❸❹. Затем используйте присваивание переноса `b`, равное `a` ❺. Обратите внимание, что нужно было привести значение к `r`, используя `std::move`. После присваивания переноса `a` находится в перенесенном состоянии и его нельзя использовать, если не переопределить его новым значением. Теперь `b` содержит сообщение, которое находилось в `a`, `We apologize for the` ❻.

Конечный продукт

Теперь мы имеем полностью реализованный `SimpleString`, который поддерживает семантику перемещения и копирования. Листинг 4.38 объединяет все это в качестве справки.

Листинг 4.38. Полностью укомплектованный класс SimpleString, поддерживающий семантику копирования и переноса

```
#include <cstdio>
#include <cstring>
#include <stdexcept>
#include <utility>

struct SimpleString {
  SimpleString(size_t max_size)
    : max_size{ max_size },
    length{} {
    if (max_size == 0) {
    throw std::runtime_error{ "Max size must be at least 1." };
    }
    buffer = new char[max_size];
    buffer[0] = 0;
  }
  ~SimpleString() {
    delete[] buffer;
  }
  SimpleString(const SimpleString& other)
    : max_size{ other.max_size },
    buffer{ new char[other.max_size] },
    length{ other.length } {
    std::strncpy(buffer, other.buffer, max_size);
  }
  SimpleString(SimpleString&& other) noexcept
    : max_size{ other.max_size },
    buffer(other.buffer),
    length(other.length) {
    other.length = 0;
    other.buffer = nullptr;
    other.max_size = 0;
  }
  SimpleString& operator=(const SimpleString& other) {
    if (this == &other) return *this;
    const auto new_buffer = new char[other.max_size];
    delete[] buffer;
    buffer = new_buffer;
    length = other.length;
    max_size = other.max_size;
    std::strncpy(buffer, other.buffer, max_size);
    return *this;
  }
  SimpleString& operator=(SimpleString&& other) noexcept {
    if (this == &other) return *this;
```

```
    delete[] buffer;
    buffer = other.buffer;
    length = other.length;
    max_size = other.max_size;
    other.buffer = nullptr;
    other.length = 0;
    other.max_size = 0;
    return *this;
  }
  void print(const char* tag) const {
    printf("%s: %s", tag, buffer);
  }
  bool append_line(const char* x) {
    const auto x_len = strlen(x);
    if (x_len + length + 2 > max_size) return false;
    std::strncpy(buffer + length, x, max_size - length);
    length += x_len;
    buffer[length++] = '\n';
    buffer[length] = 0;
    return true;
  }
private:
  size_t max_size;
  char* buffer;
  size_t length;
};
```

Методы, генерируемые компилятором

Пять методов управляют поведением переноса и копирования:

- деструктор;
- конструктор копирования;
- конструктор переноса;
- оператор присваивания копии;
- оператор присваивания переноса.

Компилятор может генерировать реализации по умолчанию для каждого из них при определенных обстоятельствах. К сожалению, правила, для которых генерируются методы, являются сложными и потенциально неравномерными в разных компиляторах.

Можно устранить эту сложность, установив для этих методов `default/delete` или применив их соответствующим образом. Это общее правило — *правило пяти*, потому что существует пять специализированных методов. Явное указание занимает немного времени, но избавляет от проблем в будущем.

В качестве альтернативы можно запомнить рис. 4.8, на котором обобщены взаимодействия между каждой из пяти реализуемых функций и каждой, которую компилятор создает от вашего имени.

Рис. 4.8. Диаграмма, показывающая, какие методы генерирует компилятор при наличии различных входных данных

Если ничего не предоставить, компилятор сгенерирует все пять функций деструктора/копирования/перемещения. Это *правило нуля*.

Если явно определить деструктор/конструктор копирования/оператор присваивания копии, все три будут сгенерированы. Это опасно, как продемонстрировано ранее для `SimpleString`: слишком легко попасть в непреднамеренную ситуацию, в которой компилятор по существу преобразует все шаги в копии.

Наконец, если предоставить только семантику переноса для класса, компилятор не будет автоматически генерировать ничего, кроме деструктора.

Итоги

Вы изучили жизненный цикл объекта. Знакомство началось с длительности хранения, где вы познакомились со временем жизни объекта от создания до уничтожения. Последующее изучение обработки исключений показало грамотную, учитывающую все случаи обработку ошибок и обогатило ваше понимание RAII. Наконец, вы увидели, как семантика копирования и переноса дает полный контроль над временем жизни объекта.

Упражнения

4.1. Создайте `struct TimerClass`. В конструкторе запишите текущее время в поле с именем `timestamp` (сравните с функцией POSIX `gettimeofday`).

4.2. В деструкторе `TimerClass` запишите текущее время и вычтите время при создании. Это время примерно равно возрасту таймера. Выведите это значение.

4.3. Реализуйте конструктор копирования и оператор присваивания копии для `TimerClass`. Копии должны иметь значения временных меток.

4.4. Реализуйте конструктор переноса и оператор присваивания переноса для `TimerClass`. Перемещенный `TimerClass` не должен выводить какие-либо выходные данные в консоль при их уничтожении.

4.5. Разработайте конструктор `TimerClass`, чтобы он принимал дополнительный параметр `const char* name`. Когда `TimerClass` уничтожается и выводит что-то с помощью стандартного вывода, добавьте имя таймера в вывод.

4.6. Поэкспериментируйте с `TimerClass`. Создайте таймер и переместите его в функцию, которая выполняет некоторые вычислительные операции (например, много математических операций в цикле). Убедитесь, что таймер ведет себя так, как ожидается.

4.7. Определите каждый метод в классе `SimpleString` (листинг 4.38). Попробуйте переопределить его с нуля, не обращаясь к книге.

Что еще почитать?

- «Оптимизация программ на C++: проверенные методы повышения производительности», Курт Гантерот (Диалектика-Вильямс, 2016)
- «Эффективный и современный C++. 42 рекомендации по использованию C++11 и C++14», Скотт Мейерс (Вильямс, 2019)

5 Полиморфизм во время выполнения

Конструктор Трурль создал однажды машину, которая умела делать все на букву «Н».

Станислав Лем, «Кибериада»

В этой главе вы узнаете, что такое полиморфизм и какие проблемы он решает. Затем вы узнаете, как добиться полиморфизма во время выполнения, который позволяет изменять поведение программ путем замены компонентов во время выполнения. Глава начинается с обсуждения нескольких важных понятий в полиморфном коде во время выполнения, включая интерфейсы, композицию объектов и наследование. Далее вы разработаете постоянно используемый пример регистрации банковских транзакций с помощью различных типов регистраторов. Глава закончится реорганизацией этого первоначального и наивного решения с помощью более элегантного подхода на основе интерфейса.

Полиморфизм

Полиморфный код — это код, который пишется один раз и который можно использовать для разных типов. В конечном счете, эта гибкость предоставляет слабосвязанный и многократно используемый код. Это устраняет утомительное копирование и вставку, делая код более понятным. C++ предлагает два полиморфных подхода. Полиморфный код во время компиляции включает полиморфные типы, которые можно определить *во время компиляции*. Другой подход — *полиморфизм во время выполнения*, который вместо этого включает типы, определенные во время выпол-

нения. Какой подход выбрать, зависит от того, знаете ли вы типы, которые нужно использовать в полиморфном коде во время компиляции или во время выполнения. Поскольку эти тесно связанные темы довольно сложны, они разделены на две главы. Глава 6 будет посвящена полиморфизму во время компиляции.

Пример для мотивации

Предположим, вы отвечаете за внедрение класса `Bank`, который переводит деньги между счетами. Аудит очень важен для транзакций класса `Bank`, поэтому с помощью класса `ConsoleLogger` поддерживается ведение журнала, как показано в листинге 5.1.

Листинг 5.1. ConsoleLogger и класс Bank, который его использует

```
#include <cstdio>

struct ConsoleLogger {
  void log_transfer(long from, long to, double amount) { ❶
    printf("%ld -> %ld: %f\n", from, to, amount); ❷
  }
};

struct Bank {
  void make_transfer(long from, long to, double amount) { ❸
    --пропуск-- ❹
    logger.log_transfer(from, to, amount); ❺
  }
  ConsoleLogger logger;
};

int main() {
  Bank bank;
  bank.make_transfer(1000, 2000, 49.95);
  bank.make_transfer(2000, 4000, 20.00);
}
--------------------------------------------------------------------------------
1000 -> 2000: 49.950000
2000 -> 4000: 20.000000
```

Во-первых, `ConsoleLogger` реализуется с помощью метода `log_transfer` ❶, который принимает детали транзакции (отправитель, получатель, сумма) и выводит их ❷. Класс `Bank` имеет метод `make_transfer` ❸, который (условно) обрабатывает транзакцию ❹, а затем использует член `logger` ❺ для регистрации транзакции. `Bank` и `ConsoleLogger` имеют отдельные проблемы – `Bank` работает с банковской логикой, а `ConsoleLogger` — с ведением журнала.

Предположим, есть требование для реализации различных типов регистраторов. Например, могут потребоваться регистратор удаленного сервера, локальный регистратор файлов или даже регистратор, который отправляет задания на принтер. Кроме того, нужно иметь возможность изменить способ регистрации программы

во время выполнения (например, администратору может потребоваться перейти от входа в систему по сети к регистрации в локальной файловой системе из-за обслуживания сервера).

Как можно выполнить такую задачу?

Простой подход заключается в использовании `enumclass` для переключения между различными регистраторами. Листинг 5.2 добавляет `FileLogger` в листинг 5.1.

Листинг 5.2. Обновленный листинг 5.1 с полиморфным регистратором во время выполнения

```
#include <cstdio>
#include <stdexcept>

struct FileLogger {
  void log_transfer(long from, long to, double amount) { ❶
    --пропуск--
    printf("[file] %ld,%ld,%f\n", from, to, amount);
  }
};

struct ConsoleLogger {
  void log_transfer(long from, long to, double amount) {
    printf("[cons] %ld -> %ld: %f\n", from, to, amount);
  }
};

enum class LoggerType { ❷
  Console,
  File
};

struct Bank {
  Bank() : type { LoggerType::Console } { } ❸

  void set_logger(LoggerType new_type) { ❹
    type = new_type;
  }

  void make_transfer(long from, long to, double amount) {
    --пропуск--
    switch(type) { ❺
    case LoggerType::Console: {
      consoleLogger.log_transfer(from, to, amount);
      break;
    } case LoggerType::File: {
      fileLogger.log_transfer(from, to, amount);
      break;
    } default: {
      throw std::logic_error("Unknown Logger type encountered.");
    } }
  }
```

```
private:
  LoggerType type;
  ConsoleLogger consoleLogger;
  FileLogger fileLogger;
};

int main() {
  Bank bank;
  bank.make_transfer(1000, 2000, 49.95);
  bank.make_transfer(2000, 4000, 20.00);
  bank.set_logger(LoggerType::File); ❻
  bank.make_transfer(3000, 2000, 75.00);
}
-------------------------------------------------------------------------------
[cons] 1000 -> 2000: 49.950000
[cons] 2000 -> 4000: 20.000000
[file] 3000,2000,75.000000
```

Вы (условно) добавили возможность регистрировать файл ❶ путем реализации `FileLogger`. Также был создан `enum class LoggerType` ❷, чтобы можно было переключать поведение при регистрации. Поле типа инициализируется для `Console` в конструкторе `Bank` ❸. В обновленном классе `Bank` добавлена функция `set_logger` ❹ для выполнения желаемого поведения при ведении журнала. `Type` используется в `make_transfer` для включения правильного регистратора ❺. Чтобы внести изменения в журнал в классе `Bank`, используется метод `set_logger` ❻ и объект обрабатывает внутреннюю диспетчеризацию.

Добавление новых регистраторов

Листинг 5.2 работает, но этот подход страдает от нескольких проблем проектирования. Добавление нового вида ведения журнала требует внесения нескольких изменений в код:

1. Необходимо написать новый тип регистратора.
2. Необходимо добавить новое значение `enum` в `enum class LoggerType`.
3. Нужно добавить новый регистр в оператор `switch` ❺.
4. Нужно добавить новый класс ведения журнала в качестве члена в `Bank`.

Слишком много работы для простого изменения!

Рассмотрим альтернативный подход, в котором `Bank` содержит указатель на регистратор. Таким образом, можно установить указатель напрямую и полностью избавиться от `LoggerType`. Вы используете тот факт, что регистраторы имеют один и тот же прототип функции. В этом и заключается идея интерфейса: классу `Bank` не нужно знать подробности реализации регистратора `Logger`, который он содержит, достаточно просто знать, как вызывать его методы.

Было бы неплохо, если бы можно было поменять `ConsoleLogger` на другой тип, который поддерживает те же операции. Скажем, `FileLogger`.

Позвольте мне представить интерфейсы.

Интерфейсы

В программной инженерии *интерфейс* — это общая граница, которая не содержит данных или кода. Он определяет сигнатуры функций, которые согласны поддерживать все реализации интерфейса. *Реализация* — это код или данные, которые декларируют поддержку интерфейса. Можно представить интерфейс как договор между классами, которые реализуют интерфейс, и пользователями (также называемыми *потребителями*) этого класса.

Потребители знают, как использовать реализации, потому что знают условия договора. Фактически потребитель никогда не должен знать базовый тип реализации. Например, в листинге 5.1 `Bank` является потребителем `ConsoleLogger`.

Интерфейсы предъявляют жесткие требования. Потребитель интерфейса может использовать только методы, явно определенные в интерфейсе. Классу `Bank` не нужно ничего знать о том, как `ConsoleLogger` выполняет свою функцию. Все, что нужно знать, это как вызвать метод `log_transfer`.

Интерфейсы поддерживают многократно используемый и слабо связанный код. Можно понять нотацию указания интерфейса, но стоит немного узнать и о композиции объектов, и о наследовании реализации.

Композиция объектов и реализация наследования

Композиция объекта — это шаблон проектирования, в котором класс содержит члены других типов классов. Альтернативный устаревший шаблон проектирования, называемый *наследованием реализации*, обеспечивает полиморфизм во время выполнения. Наследование реализации позволяет строить иерархии классов; каждый дочерний класс наследует функциональность от своих родителей. За прошедшие годы накопленный опыт наследования реализации убедил многих в том, что это антипаттерн. Например, Go и Rust — два новых и все более популярных языка системного программирования — практически не поддерживают наследование реализации. Краткое обсуждение наследования реализации оправдано по двум причинам:

- вы все еще можете столкнуться с устаревшим кодом;
- причудливый способ определения интерфейсов в C++ имеет общую линию с наследованием реализации, так что вы все равно будете знакомы с механикой.

ПРИМЕЧАНИЕ

Если вы имеете дело с перегруженным наследованием кода в C++, см. главы 20 и 21 в 4-м издании «Языка программирования C++» Бьёрна Страуструпа.

Определение интерфейсов

К сожалению, в C++ нет ключевого слова `interface`. Интерфейсы нужно определять, используя устаревшие механизмы наследования. Это только один из тех архаизмов, с которыми приходится иметь дело при работе с языком старше 40 лет. Листинг 5.3 показывает полностью определенный интерфейс `Logger` и соответствующий `ConsoleLogger`, который реализует интерфейс. По крайней мере четыре конструкции в листинге 5.3 будут вам незнакомы, и этот раздел охватывает каждую из них.

Листинг 5.3. Интерфейс Logger и переработанный ConsoleLogger

```
#include <cstdio>

struct Logger {
  virtual❶ ~Logger()❷ = default;
  virtual void log_transfer(long from, long to, double amount) = 0❸;
};

struct ConsoleLogger : Logger ❹ {
  void log_transfer(long from, long to, double amount) override ❺ {
    printf("%ld -> %ld: %f\n", from, to, amount);
  }
};
```

Для анализа листинга 5.3 необходимо понять = ключевое слово `virtual` ❶, виртуальный деструктор ❷, суффикс `=0` и чисто виртуальные методы ❸, наследование базового класса ❹ и ключевое слово `override` ❺. После этого вы узнаете, как определить интерфейс. В следующих разделах подробно обсуждаются все эти концепции.

Базовое наследование классов

В главе 4 мы узнали, что класс `exception` является базовым классом для всех других исключений `stdlib` и как классы `logic_error` и `runtime_error` наследуют от `exception`. Эти два класса, в свою очередь, образуют базовые классы для других производных классов, которые описывают условия ошибок еще более подробно, например `invalid_argument` и `system_error`. Вложенные классы исключений образуют пример иерархии классов и представляют собой дизайн наследования реализации.

Производные классы объявляются с использованием следующего синтаксиса:

```
struct DerivedClass : BaseClass {
--пропуск--
};
```

Чтобы определить отношения наследования для `DerivedClass`, используется двоеточие (`:`), за которым следует имя базового класса `BaseClass`.

Производные классы объявляются так же, как и любой другой класс. Преимущество заключается в том, что можно обрабатывать ссылки на производные классы так, как если бы они имели ссылочный тип базового класса. Листинг 5.4 использует ссылку `DerivedClass` вместо ссылки `BaseClass`.

Листинг 5.4. Программа, использующая производный класс вместо базового класса

```
struct BaseClass {}; ❶
struct DerivedClass : BaseClass {}; ❷
void are_belong_to_us(BaseClass& base) {} ❸

int main() {
  DerivedClass derived;
  are_belong_to_us(derived); ❹
}
```

`DerivedClass` ❷ является производным от `BaseClass` ❶. Функция `are_belong_to_us` принимает аргумент-ссылку на `BaseClass` — `base` ❸. Ее можно вызывать из экземпляра `DerivedClass`, потому что `DerivedClass` является производным от `BaseClass` ❹. Обратное не верно. Листинг 5.5 пытается использовать базовый класс вместо производного класса.

Листинг 5.5. Эта программа пытается использовать базовый класс вместо производного класса. (Этот листинг не будет компилироваться.)

```
struct BaseClass {}; ❶
struct DerivedClass : BaseClass {}; ❷
void all_about_that(DerivedClass& derived) {} ❸

int main() {
  BaseClass base;
  all_about_that(base); // Нет! Проблема! ❹
}
```

Здесь `BaseClass` ❶ не является производным от `DerivedClass` ❷. (Отношение наследования определяется наоборот.) Функция `all_about_that` принимает аргумент `DerivedClass` ❸. При попытке вызвать `all_about_that` с помощью `BaseClass` ❹ компилятор выдает ошибку.

Основная причина, по которой нужно наследовать класс, — это наследование его членов.

Наследование членов

Производные классы наследуют неприватные члены от своих базовых классов. Классы могут использовать унаследованные члены как обычные члены. Предполагаемое преимущество наследования членов состоит в том, что можно определить функциональность один раз в базовом классе и не повторять ее в производных классах. К сожалению, опыт убедил многих в сообществе программистов избегать наследования членов, потому что он может легко привести к хрупкому, трудному

для понимания коду по сравнению с полиморфизмом на основе композиции. (Вот почему многие современные языки программирования исключают его.)

Класс в листинге 5.6 показывает наследование членов.

Листинг 5.6. Программа, использующая унаследованные члены

```
#include <cstdio>

struct BaseClass {
  int the_answer() const { return 42; } ❶
  const char* member = "gold"; ❷
private:
  const char* holistic_detective = "Dirk Gently"; ❸
};

struct DerivedClass : BaseClass ❹ {
  void announce_agency() {
    // Эта строка не скомпилируется:
    // printf("%s's Holistic Detective Agency\n", holistic_detective); { ❼
  }
};

int main() {
  DerivedClass x;
  printf("The answer is %d\n", x.the_answer()); ❺
  printf("%s member\n", x.member); ❻
}
--------------------------------------------------------------------------------
The answer is 42 ❺
gold member ❻
```

Здесь `BaseClass` имеет публичный метод ❶, публичное поле ❷ и приватное поле ❸. `DerivedClass` объявляется как производный от `BaseClass` ❹, а затем он используется в `main`. `the_answer` ❺ и `member` ❻ доступны в `DerivedClass`, поскольку они наследуются как публичные члены. Однако раскомментирование ❼ приводит к ошибке компилятора, поскольку `holistic_detective` является закрытым и, следовательно, не наследуется производными классами.

Методы virtual

Если необходимо разрешить производному классу переопределять методы базового класса, используйте ключевое слово `virtual` (виртуальный). Добавляя `virtual` в определение метода, вы объявляете, что должна использоваться реализация в производном классе, если она предоставлена. В рамках реализации нужно добавить ключевое слово `override` в объявление метода, как показано в листинге 5.7.

`BaseClass` содержит виртуальный член ❶. В `DerivedClass` ❷ переопределяется унаследованный элемент с использованием ключевого слова `override` ❸. Реализация `BaseClass` используется только тогда, когда экземпляр `BaseClass` находится под

рукой ❹. Реализация `DerivedClass` используется, когда присутствует экземпляр `DerivedClass` ❺, даже если взаимодействие с ним осуществляется через ссылку `BaseClass` ❻.

Листинг 5.7. Программа, использующая виртуальные члены

```
#include <cstdio>

struct BaseClass {
  virtual❶ const char* final_message() const {
    return "We apologize for the incontinence.";
  }
};

struct DerivedClass : BaseClass ❷ {
  const char* final_message() const override ❸ {
    return "We apologize for the inconvenience.";
  }
};

int main() {
  BaseClass base;
  DerivedClass derived;
  BaseClass& ref = derived;
  printf("BaseClass: %s\n", base.final_message()); ❹
  printf("DerivedClass: %s\n", derived.final_message()); ❺
  printf("BaseClass&: %s\n", ref.final_message()); ❻
}
--------------------------------------------------------------------------------
BaseClass: We apologize for the incontinence. ❹
DerivedClass: We apologize for the inconvenience. ❺
BaseClass&: We apologize for the inconvenience. ❻
```

Если нужно, чтобы производный класс реализовывал метод, можно добавить суффикс `=0` к определению метода. Методы вызываются как с ключевым словом `virtual`, так и с суффиксом `=0` для чисто виртуальных методов. Нельзя создать экземпляр класса, содержащий какие-либо чисто виртуальные методы. В листинге 5.8 рассмотрим рефакторинг листинга 5.7, который использует чисто виртуальный метод в базовом классе.

Листинг 5.8. Рефакторинг листинга 5.7 с использованием чисто виртуального метода

```
#include <cstdio>

struct BaseClass {
  virtual const char* final_message() const = 0; ❶
};

struct DerivedClass : BaseClass ❷ {
  const char* final_message() const override ❸ {
    return "We apologize for the inconvenience.";
  }
};
```

```
int main() {
  // BaseClass base; // Бах! ❹
  DerivedClass derived;
  BaseClass& ref = derived;
  printf("DerivedClass: %s\n", derived.final_message()); ❺
  printf("BaseClass&: %s\n", ref.final_message()); ❻
}
--------------------------------------------------------------------------------
DerivedClass: We apologize for the inconvenience. ❺
BaseClass&: We apologize for the inconvenience. ❻
```

Суффикс `=0` указывает на чисто виртуальный метод ❶, то есть нельзя создавать экземпляр `BaseClass` — только производные от него. `DerivedClass` по-прежнему наследуется от `BaseClass` ❷, и предоставляется необходимый `final_message` ❸. Попытка создания экземпляра `BaseClass` приведет к ошибке компилятора ❹. И `DerivedClass`, и ссылка `BaseClass` ведут себя, как и раньше ❺ ❻.

ПРИМЕЧАНИЕ

Виртуальные функции могут повлечь издержки во время выполнения, хотя обычно они небольшие (в пределах 25 % от обычного вызова функции). Компилятор генерирует таблицы виртуальных функций (v-таблицы), которые содержат указатели на функции. Во время выполнения потребитель интерфейса обычно не знает его базовый тип, но он знает, как вызывать методы интерфейса (благодаря v-таблице). В некоторых случаях редактор связей может обнаруживать все виды использования интерфейса и девиртуализировать вызов функции. Это удаляет вызов функции из v-таблицы и таким образом устраняет связанные с этим затраты времени выполнения.

Чисто виртуальные классы и виртуальные деструкторы

Наследование интерфейса достигается путем наследования из базовых классов, которые содержат только чисто виртуальные методы. Такие классы называются чисто виртуальными классами. В C++ интерфейсы всегда являются *чисто виртуальными классами*. Обычно виртуальные деструкторы добавляются в интерфейсы. В некоторых редких случаях возможна утечка ресурсов, если не пометить деструкторы как виртуальные. Изучите листинг 5.9, который показывает опасность отсутствия виртуального деструктора.

Листинг 5.9. Пример, где показаны опасности отсутствия виртуальных деструкторов в базовых классах

```
#include <cstdio>

struct BaseClass {};

struct DerivedClass : BaseClass ❶ {
  DerivedClass() { ❷
    printf("DerivedClass() invoked.\n");
  }
```

```
  ~DerivedClass() { ❸
    printf("~DerivedClass() invoked.\n");
  }
};

int main() {
  printf("Constructing DerivedClass x.\n");
  BaseClass* x{ new DerivedClass{} }; ❹
  printf("Deleting x as a BaseClass*.\n");
  delete x; ❺
}
-------------------------------------------------------------------------------
Constructing DerivedClass x.
DerivedClass() invoked.
Deleting x as a BaseClass*.
```

Здесь вы видите `DerivedClass`, производный от `BaseClass` ❶. Этот класс имеет конструктор ❷ и деструктор ❸, которые выводят сообщения при вызове. Внутри `main` размещается и инициализируется `DerivedClass` с помощью `new`, а результат устанавливается в указатель `BaseClass` ❹. При удалении указателя ❺ вызывается деструктор `BaseClass`, но деструктор `DerivedClass` этого не делает!

Добавление виртуального деструктора в `BaseClass` решает проблему, как показано в листинге 5.10.

Листинг 5.10. Рефакторинг листинга 5.9 с виртуальным деструктором

```
#include <cstdio>

struct BaseClass {
  virtual ~BaseClass() = default; ❶
};

struct DerivedClass : BaseClass {
  DerivedClass() {
    printf("DerivedClass() invoked.\n");
  }
  ~DerivedClass() {
    printf("~DerivedClass() invoked.\n"); ❷
  }
};

int main() {
  printf("Constructing DerivedClass x.\n");
  BaseClass* x{ new DerivedClass{} };
  printf("Deleting x as a BaseClass*.\n");
  delete x; ❸
}
-------------------------------------------------------------------------------
Constructing DerivedClass x.
DerivedClass() invoked.
Deleting x as a BaseClass*.
~DerivedClass() invoked. ❷
```

Добавление виртуального деструктора ❶ приводит к тому, что деструктор `DerivedClass` вызывается при удалении указателя `BaseClass` ❸, в результате чего деструктор `DerivedClass` выводит сообщение ❷.

Объявление виртуального деструктора необязательно при объявлении интерфейса, но будьте осторожны. Если вы забудете, что не реализовали виртуальный деструктор в интерфейсе, и случайно сделали что-то вроде листинга 5.9, ресурсы могут быть утеряны, а компилятор не предупредит об этом.

ПРИМЕЧАНИЕ

Объявление защищенного невиртуального деструктора является хорошей альтернативой объявлению публичного виртуального деструктора, поскольку это приведет к ошибке компиляции при написании кода, удаляющего указатель базового класса. Некоторым не нравится этот подход, потому что в конечном итоге придется создать класс с публичным деструктором и при наследовании этого класса возникают те же проблемы.

Реализация интерфейсов

Чтобы объявить интерфейс, объявите чисто виртуальный класс. Чтобы реализовать интерфейс, наследуйте от него. Поскольку интерфейс является чисто виртуальным, реализация должна использовать все методы интерфейса.

Рекомендуется помечать эти методы ключевым словом `override`. Это говорит о намерении переопределить виртуальную функцию, что позволяет компилятору избавить разработчика от простых ошибок.

Использование интерфейсов

Потребитель может иметь дело только со ссылками или указателями на интерфейсы. Компилятор не может заранее знать, сколько памяти выделить для базового типа: если компилятор может знать базовый тип, лучше использовать шаблоны. Существуют два варианта настройки элемента:

- **внедрение через конструктор.** Внедрение через конструктор обычно использует ссылку на интерфейс. Поскольку ссылки не могут быть повторно установлены, они не будут меняться в течение всего срока жизни объекта;
- **внедрение через свойство.** Внедрение через свойство позволяет использовать метод для указания члена-указателя. Это позволяет изменить объект, на который направлен указатель.

Можно комбинировать эти подходы, принимая указатель интерфейса в конструкторе, а также предоставляя метод для установки указателя на что-то другое.

Как правило, вы будете использовать внедрение через конструктор, когда введенное поле не изменится в течение всего срока службы объекта. Если нужна гибкость изменения поля, предоставьте методы для внедрения через свойство.

Обновление банкового регистратора

Интерфейс `Logger` предоставляет несколько реализаций регистратора. Это позволяет потребителю `Logger` регистрировать передачи с помощью метода `log_transfer` без необходимости знать детали реализации регистратора. `ConsoleLogger` уже реализован в листинге 5.2, поэтому давайте рассмотрим, как можно добавить еще одну реализацию под названием `FileLogger`. Для простоты в этом коде изменится только префикс выходных данных журнала, но можно представить, как реализовать более сложное поведение.

Листинг 5.11 определяет `FileLogger`.

Листинг 5.11. Logger, ConsoleLogger и FileLogger

```
#include <cstdio>

struct Logger {
  virtual ~Logger() = default; ❶
  virtual void log_transfer(long from, long to, double amount) = 0; ❷
};

struct ConsoleLogger : Logger ❸ {
  void log_transfer(long from, long to, double amount) override ❹ {
    printf("[cons] %ld -> %ld: %f\n", from, to, amount);
  }
};

struct FileLogger : Logger ❺ {
  void log_transfer(long from, long to, double amount) override ❻ {
    printf("[file] %ld,%ld,%f\n", from, to, amount);
  }
};
```

`Logger` — это чисто виртуальный класс (интерфейс) с виртуальным деструктором по умолчанию ❶ и единственным методом `log_transfer` ❷. `ConsoleLogger` и `FileLogger` являются реализациями `Logger`, поскольку они — производные от интерфейса ❸❺. Вы реализовали `log_transfer` и поместили ключевое слово `override` в оба ❹❻.

Теперь рассмотрим, как можно использовать внедрение через конструктор или свойство для обновления `Bank`.

Внедрение через конструктор

Используя внедрение через конструктор, вы получаете ссылку на `Logger`, которая передается в конструктор класса `Bank`. Листинг 5.12 добавляет в листинг 5.11 включение соответствующего конструктора `Bank`. Таким образом устанавливается тип ведения журнала, который будет выполнять конкретный экземпляр `Bank`.

Листинг 5.12. Рефакторинг листинга 5.2 с использованием внедрения через конструктор, интерфейсов и композиции объектов для замены неуклюжего подхода с enum class

```
--пропуск--
// Включает листинг 5.11
struct Bank {
  Bank(Logger& logger) : logger{ logger }❶ { }
  void make_transfer(long from, long to, double amount) {
    --пропуск--
    logger.log_transfer(from, to, amount);
  }
private:
  Logger& logger;
};

int main() {
  ConsoleLogger logger;
  Bank bank{ logger }; ❷
  bank.make_transfer(1000, 2000, 49.95);
  bank.make_transfer(2000, 4000, 20.00);
}
-------------------------------------------------------------------------------
[cons] 1000 -> 2000: 49.950000
[cons] 2000 -> 4000: 20.000000
```

Конструктор класса `Bank` устанавливает значение `logger` с использованием инициализатора члена ❶. Ссылки не могут быть повторно установлены, поэтому объект, на который указывает `logger`, не изменяется в течение всего срока существования `Bank`. Выбор регистратора фиксируется при создании экземпляра `Bank` ❷.

Внедрение через свойство

Вместо использования конструктора для вставки `Logger` в `Bank` можно использовать внедрение через свойство. Этот подход использует указатель вместо ссылки. Поскольку указатели могут быть переопределены (в отличие от ссылок), можно изменить поведение `Bank` в любой момент. Листинг 5.13 является вариантом листинга 5.12 с внедрением через свойство.

Листинг 5.13. Рефакторинг листинга 5.12 с использованием внедрения свойств

```
--пропуск--
// Включает листинг 5.11

struct Bank {
  void set_logger(Logger* new_logger) {
    logger = new_logger;
  }
  void make_transfer(long from, long to, double amount) {
    if (logger) logger->log_transfer(from, to, amount);
  }
```

```
private:
  Logger* logger{};
};

int main() {
  ConsoleLogger console_logger;
  FileLogger file_logger;
  Bank bank;
  bank.set_logger(&console_logger); ❶
  bank.make_transfer(1000, 2000, 49.95); ❷
  bank.set_logger(&file_logger); ❸
  bank.make_transfer(2000, 4000, 20.00); ❹
}
--------------------------------------------------------------------------------
[cons] 1000 -> 2000: 49.950000 ❷
[file] 2000,4000,20.000000 ❹
```

Метод `set_logger` позволяет вводить новый регистратор в объект `Bank` в любой точке жизненного цикла. При задании регистратора как экземпляра `ConsoleLogger` ❶ мы получаем префикс `[cons]` в выходных данных журнала ❷. При задании регистратора как `FileLogger` ❸ мы получаем префикс `[file]` ❹.

Выбор между внедрением через конструктор или свойство

Выбор внедрения через конструктор или свойство зависит от требований к дизайну. Если важно иметь возможность изменять базовые типы элементов объекта на протяжении всего жизненного цикла объекта, следует выбрать указатели и метод внедрения через свойство. Но гибкость использования указателей и внедрения через свойство обходится дорого. В примере с банком в `Bank` необходимо убедиться, что в `logger` не было установлено значение `nullptr`, либо проверить это условие перед использованием `logger`. Существует также вопрос о поведении по умолчанию: каково начальное значение `logger`?

Одной из возможностей является предоставление выбора внедрения через конструктор и свойство. Это поощряет любого, кто использует класс, подумать об его инициализации. Листинг 5.14 показывает один из способов реализации этой стратегии.

Листинг 5.14. Рефакторинг Bank для включения внедрения через конструктор и свойство

```
#include <cstdio>

struct Logger {
  --пропуск--
};

struct Bank {
  Bank(Logger* logger) : logger{ logger } {} ❶
  void set_logger(Logger* new_logger) { ❷
    logger = new_logger;
  }
```

```
  void make_transfer(long from, long to, double amount) {
    if (logger) logger->log_transfer(from, to, amount);
  }
private:
    Logger* logger;
};
```

Как видите, можно добавить конструктор ❶ и сеттер ❷. Это требует от пользователя `Bank` инициализации регистратора со значением, даже если это просто `nullptr`. Позже пользователь может легко поменять это значение с помощью внедрения через свойство.

Итоги

В этой главе вы узнали, как определять интерфейсы, центральную роль, которую играют виртуальные функции в обеспечении работы наследования, а также некоторые общие правила использования внедрения через конструктор и свойство. Какой бы подход вы ни выбрали, комбинация наследования интерфейсов и композиции обеспечивает достаточную гибкость для большинства полиморфных приложений во время выполнения. Можно достичь безопасного типа полиморфизма во время выполнения с минимальными издержками или вообще без них. Интерфейсы поощряют инкапсуляцию и слабосвязанную конструкцию. С помощью простых сфокусированных интерфейсов можно стимулировать повторное использование кода, сделав его переносимым между проектами.

Упражнения

5.1. В `Bank` не была внедрена система бухгалтерского учета. Разработайте интерфейс под названием `AccountDatabase`, который может извлекать и устанавливать суммы на банковских счетах (идентифицируемых с помощью `long` id).

5.2. Создайте `InMemoryAccountDatabase`, реализующую `AccountDatabase`.

5.3. Добавьте ссылочный элемент `AccountDatabase` в `Bank`. Используйте внедрение через конструктор для добавления `InMemoryAccountDatabase` в `Bank`.

5.4. Измените `ConsoleLogger`, чтобы принимать `const char*` при создании. При записи в `ConsoleLogger` добавьте эту строку к выводу журнала. Обратите внимание, что можно изменить поведение при ведении журнала, не изменяя `Bank`.

Что еще почитать?

- «API Design for C++», Martin Reddy (Elsevier, 2011)

6 Полиморфизм во время компиляции

Чем больше вы адаптируетесь, тем вы интереснее.

Марта Стюарт

В этой главе вы узнаете, как добиться полиморфизма во время компиляции с помощью шаблонов. Вы узнаете, как объявлять и использовать шаблоны, обеспечивать безопасность типов, и исследуете некоторые из более сложных способов использования шаблонов. Эта глава заканчивается сравнением полиморфизма во время выполнения и компиляции в C++.

Шаблоны

C++ достигает полиморфизма во время компиляции с помощью *шаблонов*. Шаблон — это класс или функция с параметрами шаблона. Эти параметры могут заменять любой тип, включая основные и пользовательские типы. Когда компилятор видит шаблон, используемый с типом, он выделяет специально созданный экземпляр шаблона.

Создание шаблона — это процесс создания класса или функции из шаблона. Хоть это и несколько запутано, также можно ссылаться на «создание экземпляра шаблона» как на результат процесса создания шаблона. Шаблонные экземпляры иногда называют конкретными классами и конкретными типами.

Основная идея заключается в том, что вместо того, чтобы копировать и вставлять общий код повсеместно, пишется один шаблон; компилятор генерирует новые экземпляры шаблона, когда он встречает новую комбинацию типов в параметрах шаблона.

Объявление шаблонов

Шаблоны объявляются с *префиксом шаблона*, который состоит из ключевого слова `template`, за которым следуют угловые скобки `<>`. В угловых скобках размещаются объявления одного или нескольких параметров шаблона. Можно объявить параметры шаблона, используя ключевые слова `typename` или `class`, за которыми следует идентификатор. Например, префикс шаблона `template<typename T>` объявляет, что шаблон принимает параметр шаблона `T`.

ПРИМЕЧАНИЕ

Сосуществование ключевых слов `typename` и `class` является неудачным и запутанным. Они означают одно и то же. (Они оба поддерживаются по историческим причинам.) В этой главе всегда используется `typename`.

Определения класса шаблона

Рассмотрим `MyTemplateClass` в листинге 6.1, который принимает три параметра шаблона: `X`, `Y` и `Z`.

Листинг 6.1. Класс шаблона с тремя параметрами шаблона

```
template❶<typename X, typename Y, typename Z> ❷
struct MyTemplateClass❸ {
  X foo(Y&); ❹
private:
  Z* member; ❺
};
```

Префикс шаблона начинается с ключевого слова `template` ❶ и содержит параметры шаблона ❷. Эта преамбула шаблона приводит к чему-то особенному в оставшемся объявлении `MyTemplateClass` ❸. В пределах `MyTemplateClass` используются `X`, `Y` и `Z`, как если бы они были любым конкретно определенным типом, например `int` или пользовательским классом.

Метод `foo` принимает ссылку на `Y` и возвращает `X` ❹. Можно объявить элементы с типами, которые включают параметры шаблона, например указатель на `Z` ❺. Помимо специального префикса, начинающегося с ❶, этот шаблонный класс по существу идентичен нешаблонному классу.

Определения функции шаблона

Также можно указать функции шаблона, например `my_template_function` в листинге 6.2, которая также принимает три параметра шаблона: `X`, `Y` и `Z`.

В теле `my_template_function` можно использовать `arg1` и `arg2` любым удобным способом, если возвращается объект типа `X`.

Листинг 6.2. Функция шаблона с тремя параметрами

```
template<typename X, typename Y, typename Z>
X my_template_function(Y& arg1, const Z* arg2) {
  --пропуск--
}
```

Создание экземпляров шаблонов

Чтобы создать экземпляр класса шаблона, используйте следующий синтаксис:

```
tc_name❶<t_param1❷, t_param2, ...> my_concrete_class{ ... }❸;
```

`tf_name` ❶ — это место, где размещается имя класса шаблона. Затем заполняются параметры шаблона ❷. Наконец, эта комбинация имени шаблона и параметров обрабатывается, как если бы это был обычный тип: используйте любой синтаксис инициализации, который вам нравится ❸.

Создание шаблонной функции аналогично:

```
auto result = tf_name❶<t_param1❷, t_param2, ...>(f_param1❸, f_param2, ...);
```

`tf_name` ❶ — это место, где хранится имя функции шаблона. Параметры заполняются так же, как и для шаблонных классов ❷. Комбинация имени шаблона и параметров используется так же, как если бы это был обычный тип. Этот экземпляр функции шаблона вызывается через круглые скобки и параметры функции ❸.

Все эти новые обозначения могут быть пугающими для новичка, но они окажутся не так уж плохи, когда вы к ним привыкнете. Фактически это используется в наборе языковых функций, называемых именованными функциями преобразования.

Именованные функции преобразования

Именованные преобразования — это языковые функции, которые явно преобразуют один тип в другой. Именованные преобразования изредка используются в ситуациях, когда нельзя использовать неявные преобразования или конструкторы для получения нужных типов.

Все именованные преобразования принимают один параметр объекта, который нужно привести к типу (`object-to-cast`), и единственный параметр типа (`desired-type`), к которому нужно привести объект:

```
named-conversion<desired-type>(object-to-cast)
```

Например, если нужно изменить `const` объект, сначала нужно отбросить спецификатор `const`. Именованная функция преобразования `const_cast` позволяет выполнить эту операцию. Другие именованные преобразования помогают отменить неявные приведения (`static_cast`) или переинтерпретировать память с другим типом (`reinterpret_cast`).

ПРИМЕЧАНИЕ

Хотя именованные функции преобразования технически не являются шаблонными функциями, они концептуально очень близки к шаблонам — взаимосвязь отражается в их синтаксическом сходстве.

const_cast

Функция `const_cast` удаляет модификатор `const`, позволяя модифицировать значения `const`. `object-to-cast` имеет некоторый константный тип, а `desired-type` — это сам тип за вычетом квалификатора `const`.

Рассмотрим функцию `carbon_thaw` в листинге 6.3, которая принимает `const`-ссылку на аргумент `encased_solo`.

Листинг 6.3. Функция, использующая const_cast. Раскомментирование приводит к ошибке компилятора

```
void carbon_thaw(const❶ int& encased_solo) {
  //encased_solo++; ❷ // Ошибка компилятора; попытка изменения const
  auto& hibernation_sick_solo = const_cast❸<int&❹>(encased_solo❺);
  hibernation_sick_solo++; ❻
}
```

Параметр `encased_solo` имеет обозначение `const` ❶, поэтому любая попытка изменить его ❷ приведет к ошибке компилятора. `const_cast` ❸ используется для получения непостоянной ссылки `hibernation_sick_solo`. `const_cast` принимает один параметр шаблона — тип, к которому необходимо преобразовать ❹. Он также принимает параметр функции — объект, от которого нужно открепить `const` ❺. После этого можно изменять `int`, на который указывает `encased_solo`, с помощью новой непостоянной ссылки ❻.

Используйте `const_cast` только для получения доступа на запись к `const`-объектам. Любое другое преобразование типа приведет к ошибке компилятора.

ПРИМЕЧАНИЕ

Можно заведомо использовать `const_cast` для добавления `const` к типу объекта, но не нужно, так как это многословно. Вместо этого используйте неявное приведение. В главе 7 вы узнаете, что такое модификатор `volatile`. Также можно использовать `const_cast` для удаления модификатора `volatile` из объекта.

static_cast

`static_cast` отменяет четко определенное неявное преобразование, такое как приведение целочисленного типа к другому целочисленному типу. `object-to-cast` имеет некоторый тип, в который `desired-type` неявно преобразуется. Причина, по которой может понадобиться `static_cast`, заключается в том, что, как правило, неявное приведение является необратимым.

Программа в листинге 6.4 определяет функцию `increment_as_short`, которая принимает параметр указателя на `void`. Она использует `static_cast` для создания указателя на `short` из этого аргумента, увеличения указанного `short` и возврата результата. В некоторых низкоуровневых случаях применения, таких как сетевое программирование или обработка двоичных форматов файлов, может потребоваться интерпретировать необработанные байты как целочисленный тип.

Листинг 6.4. Программа, использующая static_cast

```
#include <cstdio>

short increment_as_short(void*❶ target) {
  auto as_short = static_cast❷<short*❸>(target❹);
  *as_short = *as_short + 1;
  return *as_short;
}

int main() {
  short beast{ 665 };
  auto mark_of_the_beast = increment_as_short(&beast);
  printf("%d is the mark_of_the_beast.", mark_of_the_beast);
}
-------------------------------------------------------------------------------
666 is the mark_of_the_beast.
```

Параметр `target` — это указатель на `void` ❶. `static_cast` используется для приведения `target` в `short*` ❷. Параметр шаблона — это требуемый тип ❸, а параметр функции — объект, в который нужно преобразовать ❹.

Обратите внимание, что неявное преобразование `short*` в `void*` строго определено. Попытка неточных преобразований с помощью `static_cast`, таких как преобразование `char*` в `float*`, приведет к ошибке компилятора:

```
float on = 3.5166666666;
auto not_alright = static_cast<char*>(&on); // Бах!
```

Чтобы выполнить такой опасный трюк, нужно использовать `reinterpret_cast`.

reinterpret_cast

Иногда в низкоуровневом программировании необходимо выполнять преобразования типов, которые четко не определены. В системном программировании, и особенно во встроенных средах, часто требуется полный контроль над тем, как интерпретировать память. `reinterpret_cast` предоставляет такой контроль, но обеспечение правильности этих преобразований является полностью ответственностью программиста.

Предположим, встроенное устройство хранит `unsigned long` таймер по адресу памяти 0x1000. Можно использовать `reinterpret_cast` для чтения из таймера, как показано в листинге 6.5.

Листинг 6.5. Программа, использующая reinterpret_cast. Эта программа скомпилируется, но нужно ожидать сбоя во время выполнения, если 0x1000 не читается

```
#include <cstdio>

int main() {
  auto timer = reinterpret_cast❶<const unsigned long*❷>(0x1000❸);
  printf("Timer is %lu.", *timer);
}
```

`reinterpret_cast` ❶ принимает параметр типа, соответствующий желаемому типу указателя ❷, и адрес памяти, на который результат должен указывать ❸.

Конечно, компилятор не знает, содержит ли память по адресу 0x1000 `unsigned long`. Гарантия правильности зависит только от программиста. Поскольку он берет на себя полную ответственность за эту очень опасную конструкцию, компилятор принуждает к использованию `reinterpret_cast`. Например, нельзя заменить инициализацию таймера следующей строкой:

```
const unsigned long* timer{ 0x1000 };
```

Компилятор начнет ругаться из-за преобразования `int` в указатель.

narrow_cast

Листинг 6.6 показывает пользовательский `static_cast`, который выполняет проверку выполнения на *сужение*. Сужение — это потеря информации. Подумайте о преобразовании из `int` в `short`. Пока значение `int` соответствует `short`, преобразование является обратимым и сужения не происходит. Если значение `int` слишком велико для `short`, преобразование необратимо и приводит к сужению.

Давайте реализуем именованное преобразование под названием `thin_cast`, которое проверяет сужение и выдает `runtime_error`, если оно обнаружено.

Листинг 6.6. Определение narrow_cast

```
#include <stdexcept>

template <typename To❶, typename From❷>
To❸ narrow_cast(From❹ value) {
  const auto converted = static_cast<To>(value); ❺
  const auto backwards = static_cast<From>(converted); ❻
  if (value != backwards) throw std::runtime_error{ "Narrowed!" }; ❼
  return converted; ❽
}
```

В шаблоне функции `narrow_cast` ❶ используются два параметра шаблона: тип, к которому приводится значение (`To`), и тип, из которого осуществляется преоб-

разование (`From`) ❷. Можно увидеть эти параметры шаблона в действии в качестве возвращаемого типа функции ❸ и типа значения параметра ❹. Сначала выполняется запрошенное преобразование, используя `static_cast`, чтобы получить `converted` ❺. Затем выполняется преобразование в обратном направлении (из `converted` в тип `From`), чтобы получить `backwards` ❻. Если значение не равно `backwards`, произошло сужение, поэтому выбрасывается исключение ❼. В противном случае вернется `converted` ❽.

`narrow_cast` в действии можно увидеть в листинге 6.7.

Листинг 6.7. Программа, использующая narrow_cast. (Вывод приходит из выполнения в операционной системе Windows 10 x64.)

```
#include <cstdio>
#include <stdexcept>

template <typename To, typename From>
To narrow_cast(From value) {
  --пропуск--
}
int main() {
  int perfect{ 496 }; ❶
  const auto perfect_short = narrow_cast<short>(perfect); ❷
  printf("perfect_short: %d\n", perfect_short); ❸
  try {
    int cyclic{ 142857 }; ❹
    const auto cyclic_short = narrow_cast<short>(cyclic); ❺
    printf("cyclic_short: %d\n", cyclic_short);
  } catch (const std::runtime_error& e) {
    printf("Exception: %s\n", e.what()); ❻
  }
}
--------------------------------------------------------------------------------
perfect_short: 496 ❸
Exception: Narrowed! ❻
```

Сначала `perfect` инициализируется значением 496 ❶, а затем сужается до `short-perfect_short` ❷. Это происходит без исключения, поскольку значение 496 легко вписывается в 2-байтовое сокращение в Windows 10 x64 (максимальное значение 32767). Результат получается ожидаемым ❸. Затем `cyclic` инициализируется значением 142857 ❹ и происходит попытка применить `narrow_cast` к `short cyclic_short` ❺. Это выдает `runtime_error`, потому что 142857 больше максимального значения `short` — 32767. Проверка в `narrow_cast` проваливается. Сообщение об исключении передается в `output` ❻.

Обратите внимание, что нужно предоставить только один параметр шаблона, тип возвращаемого значения, при реализации ❷❺. Компилятор может вывести параметр `From` в зависимости от использования.

mean: пример функции шаблона

Рассмотрим функцию из листинга 6.8, которая вычисляет среднее значение массива `double`, используя подход суммы и деления.

Листинг 6.8. Функция для вычисления среднего значения массива

```
#include <cstddef>

double mean(const double* values, size_t length) {
  double result{}; ❶
  for(size_t i{}; i<length; i++) {
    result += values[i]; ❷
  }
  return result / length; ❸
}
```

Переменная результата инициализируется нулем ❶. Затем значения `values` суммируются путем перебора по индексу `i` и добавления соответствующего элемента к `result` ❷. Затем `result` делится на `length` и вызывается `return` ❸.

Обобщение mean

Предположим, нужно предусмотреть вычисления `mean` для других числовых типов, таких как `float` или `long`. Можно подумать: вот для чего нужны перегрузки функций! По сути, так и есть.

Перегрузки в листинге 6.9 перегружают `mean` для поддержки массива `long`. Простой подход состоит в том, чтобы скопировать и вставить оригинал, а затем заменить экземпляры `double` на `long`.

Листинг 6.9. Перегрузка листинга 6.8, принимающая массив long

```
#include <cstddef>

long❶ mean(const long*❷ values, size_t length) {
  long result{}; ❸
  for(size_t i{}; i<length; i++) {
    result += values[i];
  }
  return result / length;
}
```

Здесь наверняка слишком много копирования и вставки ради очень небольших изменений: тип возвращаемого значения ❶, аргумент функции ❷ и `result` ❸.

Этот подход не масштабируется при добавлении новых типов. Что, если нужно поддерживать другие целочисленные типы, такие как `short` или `uint_64`? А как насчет типов `float`? Что, если позже нужно будет изменить некую логику? Предстоит много утомительного и подверженного ошибкам обслуживания.

В листинге 6.9 необходимо указать три изменения, и все они связаны с поиском и заменой типов `double` на `long`. В идеале компилятор может автоматически генерировать версии функции всякий раз, когда он сталкивается с использованием другого типа. Ключ в том, что ни одна из логик не меняется — только типы.

Для решения этой проблемы копирования и вставки понадобится универсальное программирование, стиль программирования, в котором код программируется с типами, которые еще не определены. Универсальное программирование достигается с использованием поддержки C++ для шаблонов. Шаблоны позволяют компилятору создавать пользовательский класс или функцию в зависимости от используемых типов.

Теперь, когда вы знаете, как объявлять шаблоны, снова рассмотрите функцию `mean`. По-прежнему в `mean` нужно обрабатывать широкий диапазон типов — не только `double`, — но теперь не нужно копировать и вставлять один и тот же код снова и снова.

Подумайте, как можно преобразовать листинг 6.8 в функцию шаблона, как показано в листинге 6.10.

Листинг 6.10. Рефакторинг листинга 6.8 в функцию шаблона

```
#include <cstddef>

template<typename T> ❶
T❷ mean(const T*❸ values, size_t length) {
  T❹ result{};
  for(size_t i{}; i<length; i++) {
    result += values[i];
  }
  return result / length;
}
```

Листинг 6.10 начинается с префикса шаблона ❶. Этот префикс сообщает один параметр шаблона `T`. Затем `mean` обновляется, чтобы использовать `T` вместо `double` ❷❸❹.

Теперь можно использовать `mean` со многими различными типами. Каждый раз, когда компилятор встречает использование `mean` с новым типом, он выполняет создание шаблона. Так же как при копировании, вставке и замене типов, компилятор гораздо лучше справляется с подробными монотонными задачами, чем программист. Рассмотрим пример из листинга 6.11, который вычисляет средние значения для типов `double`, `float` и `size_t`.

Листинг 6.11. Программа, использующая шаблонную функцию

```
#include <cstddef>
#include <cstdio>

template<typename T>
T mean(const T* values, size_t length) {
  --пропуск--
}

int main() {
  const double nums_d[] { 1.0, 2.0, 3.0, 4.0 };
```

```
  const auto result1 = mean<double>(nums_d, 4); ❶
  printf("double: %f\n", result1);

  const float nums_f[] { 1.0f, 2.0f, 3.0f, 4.0f };
  const auto result2 = mean<float>(nums_f, 4); ❷
  printf("float: %f\n", result2);

  const size_t nums_c[] { 1, 2, 3, 4 };
  const auto result3 = mean<size_t>(nums_c, 4); ❸
  printf("size_t: %zu\n", result3);
}
--------------------------------------------------------------------------------
double: 2.500000
float: 2.500000
size_t: 2
```

Три шаблона являются экземплярами ❶❷❸; как будто вы сгенерировали перегрузки, изолированные в листинге 6.12, вручную. (Каждый экземпляр шаблона содержит типы, выделенные жирным шрифтом, где компилятор подставил тип для параметра шаблона.)

Листинг 6.12. Реализация шаблона для листинга 6.11

```
double mean(const double* values, size_t length) {
  double result{};
  for(size_t i{}; i<length; i++) {
    result += values[i];
  }
  return result / length;
}

float mean(const float* values, size_t length) {
  float result{};
  for(size_t i{}; i<length; i++) {
    result += values[i];
  }
  return result / length;
}

char mean(const char* values, size_t length) {
  char result{};
  for(size_t i{}; i<length; i++) {
    result += values[i];
  }
  return result / length;
}
```

Компилятор проделал большую работу, но вы, возможно, заметили, что пришлось набирать указатель типа массива дважды: один раз для объявления массива и еще раз для указания параметра шаблона. Это становится утомительным и может привести к ошибкам. Если параметр шаблона не совпадает, скорее всего, это приведет к ошибке компилятора или вызовет непреднамеренное приведение.

К счастью, можно вообще пропустить параметры шаблона при вызове функции шаблона. Процесс, используемый компилятором для определения правильных параметров шаблона, называется *выводом типа шаблона*.

Вывод типа шаблона

Как правило, не нужно указывать параметры функции шаблона. Компилятор может вывести их из использования, поэтому можно переписать листинг 6.11 без них, как показано в листинге 6.13.

Листинг 6.13. Рефакторинг листинга 6.11 без явных параметров шаблона

```
#include <cstddef>
#include <cstdio>

template<typename T>
T mean(const T* values, size_t length) {
  --пропуск--
}

int main() {
  const double nums_d[] { 1.0, 2.0, 3.0, 4.0 };
  const auto result1 = mean(nums_d, 4); ❶
  printf("double: %f\n", result1);

  const float nums_f[] { 1.0f, 2.0f, 3.0f, 4.0f };
  const auto result2 = mean(nums_f, 4); ❷
  printf("float: %f\n", result2);

  const size_t nums_c[] { 1, 2, 3, 4 };
  const auto result3 = mean(nums_c, 4); ❸
  printf("size_t: %zu\n", result3);
}
--------------------------------------------------------------------------------
double: 2.500000
float: 2.500000
size_t: 2
```

Из использования ясно, что параметры шаблона — это `double` ❶, `float` ❷ и `size_t` ❸.

ПРИМЕЧАНИЕ

Вывод типа шаблона в основном работает так, как ожидается, но есть некоторый нюанс, с которым нужно ознакомиться, если вы пишете много общего кода. Для получения дополнительной информации см. Стандарт ISO [temp]. Также см. правило 1 «Эффективного использования C++» Скотта Мейерса и раздел 23.5.1 4-го издания «Программирования на языке C++» Бьёрна Страуструпа.

Иногда аргументы шаблона не могут быть выведены. Например, если возвращаемый тип функции шаблона является аргументом шаблона, полностью неза-

висимым от других функций и аргументов шаблона, аргументы шаблона должны быть указаны явно.

SimpleUniquePointer: пример класса шаблона

Уникальный указатель — это RAII-оболочка вокруг объекта, размещенного в свободном хранилище. Как следует из его названия, уникальный указатель имеет только одного владельца в одно время, поэтому, когда заканчивается срок действия уникального указателя, указанный объект уничтожается.

Тип объекта в уникальных указателях не имеет значения, что делает их главным кандидатом на класс шаблона. Рассмотрим реализацию в листинге 6.14.

Листинг 6.14. Простая реализация уникального указателя

```
template <typename T> ❶
struct SimpleUniquePointer {
  SimpleUniquePointer() = default; ❷
  SimpleUniquePointer(T* pointer)
    : pointer{ pointer } { ❸
  }
  ~SimpleUniquePointer() { ❹
    if(pointer) delete pointer;
  }
  SimpleUniquePointer(const SimpleUniquePointer&) = delete;
  SimpleUniquePointer& operator=(const SimpleUniquePointer&) = delete; ❺
  SimpleUniquePointer(SimpleUniquePointer&& other) noexcept ❻
    : pointer{ other.pointer } {
    other.pointer = nullptr;
  }
  SimpleUniquePointer& operator=(SimpleUniquePointer&& other) noexcept { ❼
    if(pointer) delete pointer;
    pointer = other.pointer;
    other.pointer = nullptr;
    return *this;
  }
  T* get() { ❽
    return pointer;
  }
private:
  T* pointer;
};
```

Класс шаблона объявляется с префиксом шаблона ❶, который устанавливает `T` как тип обернутого объекта. Затем указывается конструктор по умолчанию, используя ключевое слово `default` ❷. (Вспомните из главы 4, что стандарт по умолчанию нужен, когда необходимо использовать и конструктор по умолчанию, и конструктор не по умолчанию.) Сгенерированный конструктор по умолчанию установит для приватного указателя `T*` значение `nullptr` благодаря правилам инициализации по

умолчанию. Существует и конструктор не по умолчанию, который принимает `T*` и устанавливает указатель на приватный член ❸. Поскольку указатель, возможно, равен `nullptr`, деструктор проверяет это перед удалением ❹.

Поскольку нужно разрешить использование только одного владельца указанного объекта, конструктор копирования и оператор присваивания копии удаляются ❺. Это предотвращает проблемы двойного освобождения, которые обсуждались в главе 4. Однако можно сделать уникальный указатель перемещаемым, добавив конструктор переноса ❻. Это крадет значение `pointer` из `other`, а затем устанавливает указатель `other` на `nullptr`, передавая ему ответственность объекта, на который указывает указатель. Как только конструктор переноса завершает работу, перемещенный объект уничтожается. Поскольку указатель перемещенного объекта установлен на `nullptr`, деструктор не будет удалять указанный объект.

Возможность того, что `this` уже владеет объектом, усложняет присваивание переноса ❼. Нужно явно проверить предыдущее владение, потому что неудача в удалении указателя приводит к утечке ресурсов. После этой проверки выполняются те же операции, что и в конструкторе копирования: указатель задается значением `other.pointer`, а затем `other.pointer` устанавливается на значение `nullptr`. Это гарантирует, что перемещенный объект не удалит указанный объект.

Можно получить прямой доступ к базовому указателю, вызвав метод `get` ❽.

Давайте подключим нашего старого друга `Tracer` из листинга 4.5 для исследования `SimpleUniquePointer`. Рассмотрите программу в листинге 6.15.

Листинг 6.15. Программа, исследующая SimpleUniquePointers с классом Tracer

```
#include <cstdio>
#include <utility>

template <typename T>
struct SimpleUniquePointer {
  --пропуск--
};

struct Tracer {
  Tracer(const char* name) : name{ name } {
    printf("%s constructed.\n", name); ❶
  }
  ~Tracer() {
    printf("%s destructed.\n", name); ❷
  }
private:
  const char* const name;
};

void consumer(SimpleUniquePointer<Tracer> consumer_ptr) {
  printf("(cons) consumer_ptr: 0x%p\n", consumer_ptr.get()); ❸
}
```

```
int main() {
  auto ptr_a = SimpleUniquePointer(new Tracer{ "ptr_a" });
  printf("(main) ptr_a: 0x%p\n", ptr_a.get()); ❹
  consumer(std::move(ptr_a));
  printf("(main) ptr_a: 0x%p\n", ptr_a.get()); ❺
}
--------------------------------------------------------------------------------
ptr_a constructed. ❶
(main) ptr_a: 0x000001936B5A2970 ❹
(cons) consumer_ptr: 0x000001936B5A2970 ❸
ptr_a destructed. ❷
(main) ptr_a: 0x0000000000000000 ❺
```

Во-первых, `Tracer` динамически выделяется с сообщением `ptr_a`. Это выводит первое сообщение ❶. Полученный указатель `Tracer` используется для создания `SimpleUniquePointer` с именем `ptr_a`. Далее используется метод `get()` из `ptr_a` для получения адреса его `Tracer`, который также выводится в консоль ❹. Затем используется `std::move` для освобождения `Tracer ptr_a` от функции `consumer`, которая перемещает `ptr_a` в аргумент `consumer_ptr`.

Теперь `consumer_ptr` принадлежит `Tracer`. метод `get()`в `consumer_ptr` используется для получения адреса `Tracer`, а затем выводится возвращаемое значение ❸. Обратите внимание, что этот адрес совпадает с адресом, выведенным в ❹. Когда `consumer` возвращает значение, `consumer_ptr` уничтожается, потому что его длительность хранения является областью действия `consumer`. В результате `ptr_a` уничтожается ❷.

Вспомните, что `ptr_a` находится в перемещенном состоянии — его `Tracer` был перемещен в `consumer`. Метод `get()` в `ptr_a` используется, чтобы показать, что теперь он содержит `nullptr` ❺.

Благодаря `SimpleUniquePointer` вы пропустите динамически размещенный объект; кроме того, поскольку `SimpleUniquePointer` просто переносит указатель под капот, семантика переноса оказывается эффективной.

ПРИМЕЧАНИЕ

`SimpleUniquePointer` — это педагогическая реализация `std::unique_ptr` из stdlib, которая является членом семейства шаблонов RAII, называемых умными указателями. Вы узнаете об этом в части 2.

Проверка типов в шаблонах

Шаблоны безопасны. Во время создания шаблона компилятор вставляет параметры шаблона. Если полученный код неверен, компилятор не будет генерировать экземпляр.

Рассмотрим функцию шаблона из листинга 6.16, которая возводит элемент в квадрат и возвращает результат.

Листинг 6.16. Функция шаблона, которая возводит значение в квадрат

```
template<typename T>
T square(T value) {
  return value * value; ❶
}
```

T имеет молчаливое требование: он должен поддерживать умножение ❶.

Если вы попытаетесь использовать `square` с, скажем, `char*`, компиляция завершится неудачно, как показано в листинге 6.17.

Листинг 6.17. Программа с ошибочным созданием шаблона. (Эта программа не компилируется.)

```
template<typename T>
T square(T value) {
  return value * value;
}

int main() {
  char my_char{ 'Q' };
  auto result = square(&my_char); ❶ // Бах!
}
```

Указатели не поддерживают умножение, поэтому инициализация шаблона не удалась ❶.

Функция возведения в квадрат заведомо мала, но сообщение об ошибке при инициализации неудачного шаблона не отображается. На MSVC v141 получится это:

```
main.cpp(3): error C2296: '*': illegal, left operand has type 'char *'
main.cpp(8): note: see reference to function template instantiation 'T
*square<char*>(T)' being compiled
        with
        [
            T=char *
        ]
main.cpp(3): error C2297: '*': illegal, right operand has type 'char *'
```

На GCC 7.3 — это:

```
main.cpp: In instantiation of 'T square(T) [with T = char*]':
main.cpp:8:32: required from here
main.cpp:3:16: error: invalid operands of types 'char*' and 'char*' to binary
'operator*'
   return value * value;
          ~~~~~^~~~~~
```

Такие сообщения являются ярким примером весьма загадочных сообщений об ошибках, возникающих при сбоях инициализации шаблона.

Хотя создание шаблона обеспечивает безопасность типов, проверка происходит очень поздно в процессе компиляции. Когда компилятор создает экземпляр шаблона, он вставляет типы параметров шаблона в шаблон. После вставки типа компилятор

пытается скомпилировать результат. Если создание экземпляра завершается неудачей, компилятор выдает слова умирающего внутри экземпляра шаблона.

Шаблонное программирование на C++ имеет сходство с *языками с утиной типизацией*. Языки с утиной типизацией (например, Python) откладывают проверку типов до времени выполнения. Суть в том, что если объект выглядит как утка и крякает как утка, то это утка. К сожалению, это значит, что обычно нельзя знать, поддерживает ли объект определенную операцию, до выполнения программы.

С шаблонами невозможно узнать, удастся ли создать экземпляр до попытки его компиляции. Хотя языки с утиной типизацией могут аварийно завершиться во время выполнения, шаблоны могут завершиться во время компиляции.

Здравомыслящие люди в сообществе C++ считают такую ситуацию неприемлемой, поэтому есть блестящее решение — концепты.

Концепты

Концепты ограничивают параметры шаблона, позволяя проверять параметры в момент создания, а не в момент первого использования. Улавливая проблемы использования в момент создания экземпляра, компилятор может выдать дружественный и информативный код ошибки, например: «Вы пытались создать экземпляр этого шаблона с помощью `char*`, но для этого шаблона требуется тип, поддерживающий умножение».

Концепты позволяют выражать требования к параметрам шаблона непосредственно на языке.

К сожалению, концепты еще официально не являются частью стандарта C++, хотя за них проголосовали в C++ 20. На момент публикации GCC 6.0 и более поздние версии поддерживают Техническую спецификацию концептов и Microsoft активно работает над реализацией концептов в своем компиляторе C++, MSVC. Независимо от их неофициального статуса стоит изучить концепты более подробно по нескольким причинам:

- они в корне изменят способ, которым достигается полиморфизм во время компиляции. Знакомство с концепциями принесет большие дивиденды;
- они предоставляют концептуальную основу для понимания некоторых временных решений, которые можно использовать для обработки ошибок компиляции при неправильном использовании шаблонов;
- они обеспечивают отличный концептуальный мост между шаблонами времени компиляции и интерфейсами, основным механизмом полиморфизма во время выполнения (рассматривается в главе 5);
- если вы можете использовать GCC 6.0 или более позднюю версию, концепты доступны, если включить флаг компилятора `-fconcepts`.

ПРЕДУПРЕЖДЕНИЕ

Окончательная спецификация концепции C++ 20 почти наверняка будет отличаться от Технической спецификации концептов. В этом разделе представлены концепты, как указано в Технической спецификации концептов, чтобы вы могли ей следовать.

Определение концепта

Концепт — это шаблон. Это постоянное выражение, включающее аргументы шаблона, вычисляемое во время компиляции. Представьте концепт как один большой *предикат*: функцию, которая вычисляет `true` или `false`.

Если набор параметров шаблона соответствует критериям для данного концепта, этот концепт оценивается как `true`, когда создается экземпляр с данными параметрами; в противном случае он будет оценен как `false`. Когда концепт оценивается как `false`, создание экземпляра шаблона завершается неудачно.

Концепты объявляются, используя ключевое слово `concept` в другом хорошо известном определении функции шаблона:

```
template<typename T1, typename T2, ...>
concept bool ConceptName() {
  --пропуск--
}
```

Типажи типа

Концепты проверяют параметры типа. Внутри концепций происходит управление типами для проверки их свойств. Эти манипуляции можно выполнить вручную или использовать библиотеку поддержки типов, встроенную в stdlib. Библиотека содержит утилиты для проверки свойств типов. Эти утилиты в совокупности называются *типажами типа*. Они доступны в заголовке `<type_traits>` и являются частью пространства имен `std`. Таблица 6.1 перечисляет некоторые часто используемые типажи типа.

ПРИМЕЧАНИЕ

См. раздел 5.4 второго издания «Стандартной библиотеки C++» Николая М. Джосаттиса, где приведен исчерпывающий список типажей типов, доступных в stdlib.

Каждый типаж типа является классом шаблона, который принимает один параметр шаблона, тип, который нужно проверить. Результаты извлекаются при помощи статического члена шаблона `value`. Этот член равен `true`, если параметр типа соответствует критериям; в противном случае он равен `false`.

Рассмотрим классы типажей типа `is_integral` и `is_floating_point`. Они полезны для проверки, является ли тип (как вы уже догадались) целочисленным или с плавающей

точкой. Оба шаблона принимают один параметр шаблона. Пример в листинге 6.18 исследует типажи типа с несколькими типами.

Таблица 6.1. Выбранные типажи типа из заголовка `<type_traits>`

Типаж типа	Проверяет, является ли параметр шаблона...
`is_void`	`void`
`is_null_pointer`	`nullptr`
`is_integral`	`bool`, тип `char`, тип `int`, тип `short`, тип `long`, тип `long long`
`is_floating_point`	`float`, `double`, или `long double`
`is_fundamental`	Любой из `is_void`, `is_null_pointer`, `is_integral`, или `is_floating_point`
`is_array`	Массив; это тип, содержащий квадратные скобки []
`is_enum`	Тип перечисления (`enum`)
`is_class`	Тип `class` (но не `union`)
`is_function`	Функция
`is_pointer`	Указатель; учитываются указатели на функции, но не на члены классов и `nullptr`
`is_reference`	Ссылка (на `l`-значение или `r`-значение)
`is_arithmetic`	`is_floating_point` или `is_integral`
`is_pod`	Тип простых данных; это тип данных, который может быть представлен в обычном C
`is_default_constructible`	Конструируется по умолчанию; то есть он может быть построен без аргументов или значений инициализации
`is_constructible`	Создается с предоставленными параметрами шаблона: этот типаж типа позволяет пользователю предоставлять дополнительные параметры шаблона помимо рассматриваемого типа
`is_copy_constructible`	Конструируется копированием
`is_move_constructible`	Конструируется переносом
`is_destructible`	Уничтожаем
`is_same`	Тот же тип, что и у дополнительного параметра шаблона (включая модификаторы `const` и `volatile`)
`is_invocable`	Вызывается с заданными параметрами шаблона: этот типаж типа позволяет пользователю предоставлять дополнительные параметры шаблона помимо рассматриваемого типа

Листинг 6.18. Программа, использующая типажи типа

```
#include <type_traits>
#include <cstdio>
#include <cstdint>

constexpr const char* as_str(bool x) { return x ? "True" : "False"; } ❶

int main() {
  printf("%s\n", as_str(std::is_integral<int>::value)); ❷
  printf("%s\n", as_str(std::is_integral<const int>::value)); ❸
  printf("%s\n", as_str(std::is_integral<char>::value)); ❹
  printf("%s\n", as_str(std::is_integral<uint64_t>::value)); ❺
  printf("%s\n", as_str(std::is_integral<int&>::value)); ❻
  printf("%s\n", as_str(std::is_integral<int*>::value)); ❼
  printf("%s\n", as_str(std::is_integral<float>::value)); ❽
}
--------------------------------------------------------------------------------
True ❷
True ❸
True ❹
True ❺
False ❻
False ❼
False ❽
```

Листинг 6.18 определяет вспомогательную функцию `as_str` ❶ для вывода логических значений со строками `True` или `False`. В `main` выводится результат создания различных типажей типов. Параметры шаблона `int` ❷, `constint` ❸, `char` ❹ и `uint64_t` ❺ возвращают `true` при передаче в `is_integral`. Ссылочные типы ❻❼ и типы с плавающей точкой ❽ возвращают `false`.

ПРИМЕЧАНИЕ

Вспомните, что `printf` не имеет спецификатора формата для `bool`. Вместо использования целочисленного спецификатора формата `%d` в качестве замены в листинге 6.18 используется функция `as_str`, которая возвращает строковый литерал `True` или `False` в зависимости от значения `bool`. Поскольку эти значения являются строковыми литералами, их можно использовать с заглавными буквами.

Типажи типа часто являются строительными блоками для концепта, но иногда нужно больше гибкости. Типажи типа говорят, *что* это за типы, но иногда необходимо также указывать, *как* шаблон будет их использовать. Для этого используются требования.

Требования

Требования — это специальные ограничения для параметров шаблона. Каждый концепт может указывать любое количество требований к параметрам своего шаблона.

Требования закодированы в выражения требований, обозначенные ключевым словом `requires`, за которым следуют аргументы функции и тело.

Последовательность синтаксических требований включает в себя тело выражения требований. Каждое синтаксическое требование накладывает ограничение на параметры шаблона. В совокупности выражения обязательно имеют следующую форму:

```
requires (аргумент-1, аргумент-2, ...❶) {
  { выражение1❷ } -> возвращаемый-тип1❸;
  { выражение2 } -> возвращаемый-тип2;
  --пропуск--
}
```

Выражения требований принимают параметры, которые ставятся после ключевого слова `requires` ❶. Эти параметры имеют типы, полученные из параметров шаблона.

Далее следуют синтаксические требования, каждое из которых обозначено `{} ->`. Произвольное выражение помещается в каждую из фигурных скобок ❷. Это выражение может включать любое количество параметров в выражении параметров.

Если создание экземпляра приводит к тому, что синтаксическое выражение не компилируется, это синтаксическое требование не выполняется. Предположим, что выражение вычисляется без ошибок, следующая проверка — соответствует ли тип возвращаемого значения этого выражения типу, указанному после стрелки `->` ❸. Если вычисленный тип результата выражения ❸ не может неявно преобразоваться в возвращаемый тип, синтаксическое требование не выполняется.

Если какое-либо из синтаксических требований не выполняется, выражение требований вычисляется как `false`. Если все синтаксические требования выполнены, выражение требований вычисляется как `true`.

Предположим, есть два типа, `T` и `U`, и необходимо узнать, можно ли сравнивать объекты этих типов, используя операторы равенства `==` и неравенства `!=`. Одним из способов кодирования этого требования является использование следующего выражения:

```
// T, U — это типы
requires (T t, U u) {
  { t == u } -> bool; // синтаксическое требование 1
  { u == t } -> bool; // синтаксическое требование 2
  { t != u } -> bool; // синтаксическое требование 3
  { u != t } -> bool; // синтаксическое требование 4
}
```

Выражение требований принимает два параметра, каждый из которых имеет типы `T` и `U`.

Каждое из синтаксических требований, содержащихся в выражении требований, является выражением, использующим `t` и `u` с `==` или `!=`. Все четыре синтаксических требования приводят к результату типа `bool`. Любые два типа, удовлетворяющие этому требованию, гарантированно поддерживают сравнение с `==` и `!=`.

Создание концепта из выражений требования

Поскольку выражения требуют вычисления во время компиляции, концепты могут содержать любое их количество. Попробуйте создать концепт, который защищает от злоупотребления `mean`. В листинге 6.19 приведены некоторые неявные требования, использованные ранее в листинге 6.10.

Листинг 6.19. Листинг 6.10 с аннотациями для некоторых неявных требований к T

```
template<typename T>
T mean(T* values, size_t length) {
  T result{}; ❶
  for(size_t i{}; i<length; i++) {
    result ❷+= values[i];
  }
  ❸return result / length;
}
```

Можно заметить три требования, подразумеваемые этим кодом:

- `T` должен быть конструируемым по умолчанию ❶;
- `T` поддерживает оператор `+=` ❷;
- деление `T` на `size_t` приводит к `T` ❸.

Из этих требований можно создать концепт под названием `Averageable`, как показано в листинге 6.20.

Листинг 6.20. Концепт Averageable. Аннотации соответствуют требованиям и телу mean

```
template<typename T>
concept bool Averageable() {
  return std::is_default_constructible<T>::value ❶
    && requires (T a, T b) {
      { a += b } -> T; ❷
      { a / size_t{ 1 } } -> T; ❸
    };
}
```

Типаж типа `is_default_constructible` используется, чтобы убедиться, что `T` является конструируемым по умолчанию ❶, что можно добавить два типа `T` ❷ и что можно разделить `T` на `size_t` ❸ и получить результат типа `T`.

Напомним, что концепты — это просто предикаты; вы создаете логическое выражение, которое оценивается как `true`, когда параметры шаблона поддерживаются, и `false`, если они не поддерживаются. Концепт состоит из типажа типа ❶ и ожидает наличия двух выражений требований ❷❸. Если любой из трех возвращает `false`, ограничения концепта не выполняются.

Использование концептов

Объявление концептов занимает гораздо больше работы, чем их использование. Чтобы использовать концепт, просто используйте имя концепта вместо ключевого слова `typename`.

Например, можно переписать листинг 6.13 с помощью концепта `Averageable`, как показано в листинге 6.21.

Листинг 6.21. Рефакторинг листинга 6.13 с использованием Averageable

```
#include <cstddef>
#include <type_traits>

template<typename T>
concept bool Averageable() { ❶
  --пропуск--
}

template<Averageable❷ T>
T mean(const T* values, size_t length) {
  --пропуск--
}

int main() {
  const double nums_d[] { 1.0f, 2.0f, 3.0f, 4.0f };
  const auto result1 = mean(nums_d, 4);
  printf("double: %f\n", result1);
  const float nums_f[] { 1.0, 2.0, 3.0, 4.0 };
  const auto result2 = mean(nums_f, 4);
  printf("float: %f\n", result2);
  const size_t nums_c[] { 1, 2, 3, 4 };
  const auto result3 = mean(nums_c, 4);
  printf("size_t: %d\n", result3);
}
--------------------------------------------------------------------------------
double: 2.500000
float: 2.500000
size_t: 2
```

После определения `Averageable` ❶ вы просто используете его вместо `typename` ❷. Никаких дальнейших изменений не требуется. Код, сгенерированный при компиляции листинга 6.13, идентичен коду, сгенерированному при компиляции листинга 6.21.

Использование концептов окупается: при попытке использовать `mean` с типом, который не является `Averageable`, вы получите ошибку компилятора в момент создания. Это предоставляет гораздо лучшие сообщения об ошибках компилятора, чем те, что выдаются из необработанного шаблона.

Посмотрите на экземпляр `mean` в листинге 6.22, где вы «случайно» пытаетесь усреднить массив указателей `double`.

Листинг 6.22. Неправильная реализация шаблона с использованием не-Averageable параметра

```
--пропуск--
int main() {
  auto value1 = 0.0;
  auto value2 = 1.0;
  const double* values[] { &value1, &value2 };
  mean(values❶, 2);
}
```

Есть несколько проблем с использованием `values` ❶. Что компилятор может сказать об этих проблемах?

Без концептов GCC 6.3 выдает сообщение об ошибке, показанное в листинге 6.23.

Листинг 6.23. Сообщение об ошибке из GCC 6.3 при компиляции листинга 6.22

```
<source>: In instantiation of 'T mean(const T*, size_t) [with T = const
double*; size_t = long unsigned int]':
<source>:17:17: required from here
<source>:8:12: error: invalid operands of types 'const double*' and 'const
double*' to binary 'operator+'
     result += values[i]; ❶
     ~~~~~~~^~~~~~~~~~
<source>:8:12: error:   in evaluation of 'operator+=(const double*, const
double*)'
<source>:10:17: error: invalid operands of types 'const double*' and 'size_t'
{aka 'long unsigned int'} to binary 'operator/'
   return result / length; ❷
   ~~~~~~~^~~~~~~~
```

Можно ожидать, что это сообщение об ошибке будет крайне смущать случайного пользователя `mean`. Что такое `i` ❶ ? Почему `const double*` участвует в делении ❷ ?

Концепты предоставляют гораздо более наглядное сообщение об ошибке, как показано в листинге 6.24.

Листинг 6.24. Сообщение об ошибке из GCC 7.2 при компиляции листинга 6.22 с включенными концептами

```
<source>: In function 'int main()':
<source>:28:17: error: cannot call function 'T mean(const T*, size_t) [with T
= const double*; size_t = long unsigned int]'
   mean(values, 2); ❶
                ^
<source>:16:3: note: constraints not satisfied
 T mean(const T* values, size_t length) {
   ^~~~
<source>:6:14: note: within 'template<class T> concept bool Averageable()
[with T = const double*]'
 concept bool Averageable() {
              ^~~~~~~~~~
<source>:6:14: note:     with 'const double* a'
<source>:6:14: note:     with 'const double* b'
<source>:6:14: note: the required expression '(a + b)' would be ill-formed ❷
<source>:6:14: note: the required expression '(a / b)' would be ill-formed ❸
```

Это сообщение об ошибке просто фантастическое. Компилятор сообщает, какой аргумент (`values`) не соответствует ограничению ❶. Затем он сообщает, что значения не являются `Averageable`, поскольку они не удовлетворяют двум обязательным выражениям ❷ ❸. Вы сразу же знаете, как изменить параметры, чтобы сделать создание этого шаблона успешным.

После включения концептов в стандарт C++ вполне вероятно, что stdlib будет включать в себя много концептов. Целью разработки концептов является то, что программисту не нужно самостоятельно определять очень много концептов; скорее они должны быть в состоянии объединить концепты и специальные требования в префиксе шаблона. В табл. 6.2 приведен неполный список некоторых концептов, которые вы, возможно, захотите добавить; они заимствованы из реализации концепций Эндрю Саттона в библиотеке Origins.

ПРИМЕЧАНИЕ

См. github.com/asutton/origin/ для получения дополнительной информации о библиотеке Origins. Чтобы скомпилировать приведенные ниже примеры, можно установить Origins и использовать GCC версии 6.0 или новее с флагом `-fconcepts`.

Таблица 6.2. Концепты, содержащиеся в библиотеке Origins

Концепт	Тип, который...
`Conditional`	Может быть явно приведен к `bool`
`Boolean`	Является `Conditional` и поддерживает логические операции `!`, `&&` и `||`
`Equality_comparable`	Поддерживает операции `==` и `!=`, которые возвращают логическое значение
`Destructible`	Может быть уничтожен (сравните с `is_destructible`)
`Movable`	Поддерживает семантику переноса: он должен поддерживать конструктор переноса и присваивание переноса (сравните с `is_move_assignable` и `is_move_constructible`)
`Copyable`	Поддерживает семантику копирования: он должен поддерживать конструктор копирования и присваивание копии (сравните с `is_copy_constructible` и `is_copy_assignable`)
`Regular`	Является по умолчанию конструируемым, копируемым и `Equality_comparable`
`Ordered`	Является `Regular` и полностью упорядочен (и может использоваться в сортировке)
`Number`	Является `Ordered` и поддерживает математические операции вроде `+`, `–`, `/` и `*`
`Function`	Поддерживает вызовы; то есть его можно вызывать (сравните с `is_invocable`)
`Predicate`	Является `Function` и возвращает `bool`
`Range`	Может использоваться в циклах `for` на основе диапазона

Есть несколько способов встроить ограничения в префикс шаблона. Если параметр шаблона используется только для объявления типа параметра функции, можно полностью пропустить префикс шаблона:

```
возвращаемый-тип имя-функции(Концепт1❶ параметр-1, ...) {
  --пропуск--
}
```

Поскольку используется концепт, а не `typename` для определения типа параметра ❶, компилятор знает, что связанная функция является шаблоном. Можно даже смешивать концепты и конкретные типы в списке параметров. Другими словами, всякий раз при использовании концепта как части определения функции эта функция становится шаблоном.

Функция шаблона в листинге 6.25 принимает массив элементов `Ordered` и находит минимальный из них.

Листинг 6.25. Функция шаблона, использующая концепт Ordered

```
#include <origin/core/concepts.hpp>
size_t index_of_minimum(Ordered❶* x, size_t length) {
  size_t min_index{};
  for(size_t i{ 1 }; i<length; i++) {
    if(x[i] < x[min_index]) min_index = i;
  }
  return min_index;
}
```

Несмотря на отсутствие префикса шаблона, `index_of_minimum` — это шаблон, потому что `Ordered` — это концепт ❶. Этот шаблон может быть создан так же, как и любая другая функция шаблона, как показано в листинге 6.26.

Листинг 6.26. Листинг, использующий index_of_minimum из листинга 6.25. Раскомментирование приводит к сбою компиляции

```
#include <cstdio>
#include <cstdint>
#include <origin/core/concepts.hpp>

struct Goblin{};

size_t index_of_minimum(Ordered* x, size_t length) {
  --пропуск--
}

int main() {
  int x1[] { -20, 0, 100, 400, -21, 5123 };
  printf("%zu\n", index_of_minimum(x1, 6)); ❶
  unsigned short x2[] { 42, 51, 900, 400 };
  printf("%zu\n", index_of_minimum(x2, 4)); ❷
  Goblin x3[] { Goblin{}, Goblin{} };
  //index_of_minimum(x3, 2); ❸ // Бах! Goblin не является Ordered.
}
--------------------------------------------------------------------------------
4 ❶
0 ❷
```

Создание экземпляров для массивов `int` ❶ и `unsigned short` ❷ выполняется успешно, поскольку типы являются `Ordered` (см. табл. 6.2).

Однако класс `Goblin` не `Ordered`, и создание экземпляра шаблона завершится неудачно при попытке компиляции ❸. Важно, что сообщение об ошибке будет информативным:

```
error: cannot call function 'size_t index_
of_minimum(auto:1*, size_t) [with auto:1 = Goblin; size_t = long unsigned int]'
   index_of_minimum(x3, 2); // Бах! Goblin is not Ordered.
                           ^
note: constraints not satisfied
 size_t index_of_minimum(Ordered* x, size_t length) {
        ^~~~~~~~~~~~~~~~
note: within 'template<class T> concept bool origin::Ordered() [with T =
Goblin]'
 Ordered()
```

Вы знаете, что не удалось создать экземпляр `index_of_minimum` и проблема связана с концептом `Ordered`.

Специальные выражения требований

Концепты — это довольно тяжелые механизмы обеспечения безопасности типов. Иногда просто нужно применить некоторые требования непосредственно в префиксе шаблона. Для достижения этой цели можно встраивать необходимые выражения непосредственно в определение шаблона. Рассмотрим функцию `get_copy` в листинге 6.27, которая принимает указатель и безопасно возвращает копию указанного объекта.

Листинг 6.27. Функция шаблона со специальным выражением требования

```
#include <stdexcept>

template<typename T>
  requires❶ is_copy_constructible<T>::value ❷
T get_copy(T* pointer) {
  if (!pointer) throw std::runtime_error{ "Null-pointer dereference" };
  return *pointer;
}
```

Префикс шаблона содержит ключевое слово `requires` ❶, с которого начинается выражение требования. В этом случае типаж типа `is_copy_constructible` гарантирует, что `T` является копируемым ❷. Таким образом, если пользователь случайно попытается получить `get_copy` с указателем, указывающим на некопируемый объект, ему будет предоставлено четкое объяснение того, почему не удалось создать экземпляр шаблона. Рассмотрим пример в листинге 6.28.

За определением `get_copy` ❶ следует определение класса `Highlander`, которое содержит конструктор по умолчанию ❷ и удаленный конструктор копирования ❸. В `main` вы инициализировали `Highlander` ❹, взяли ссылку на него ❺ и попытались

создать `get_copy` с результатом ❻. Поскольку может существовать только один `Highlander` (он не подлежит копированию), в листинге 6.28 выдается совершенно четкое сообщение об ошибке.

Листинг 6.28. Программирование с использованием шаблона get_copy из листинга 6.27. Этот код не компилируется

```
#include <stdexcept>
#include <type_traits>

template<typename T>
  requires std::is_copy_constructible<T>::value
T get_copy(T* pointer) { ❶
  --пропуск--
}

struct Highlander {
  Highlander() = default; ❷
  Highlander(const Highlander&) = delete; ❸
};

int main() {
  Highlander connor; ❹
  auto connor_ptr = &connor; ❺
  auto connor_copy = get_copy(connor_ptr); ❻
}
--------------------------------------------------------------------------------
In function 'int main()':
error: cannot call function 'T get_copy(T*) [with T = Highlander]'
   auto connor_copy = get_copy(connor_ptr);
                                          ^
note: constraints not satisfied
 T get_copy(T* pointer) {
   ^~~~~~~
note: 'std::is_copy_constructible::value' evaluated to false
```

static_assert: полумеры до концептов

Начиная с C++ 17, концепты не являются частью стандарта, поэтому доступны не для всех компиляторов. Между этим можно применить временную задержку: выражение `static_assert`. Эти выражения вычисляются во время компиляции. Если утверждение не выполняется, компилятор выдаст ошибку и при необходимости предоставит диагностическое сообщение. `static_assert` имеет следующую форму:

```
static_assert(логическое-выражение, необязательное-сообщение);
```

При отсутствии концептов можно добавить одно или несколько выражений `static_assert` в тела шаблонов, чтобы помочь пользователям в диагностике ошибок использования.

Предположим, нужно улучшить сообщения об ошибках в `mean`, не опираясь на концепты. Можно использовать типажи типа в сочетании со `static_assert` для достижения аналогичного результата, как показано в листинге 6.29.

Листинг 6.29. Использование выражений static_assert для исправления ошибок во время компиляции в mean в листинге 6.10

```
#include <type_traits>

template <typename T>
T mean(T* values, size_t length) {
  static_assert(std::is_default_constructible<T>(),
    "Type must be default constructible."); ❶
  static_assert(std::is_copy_constructible<T>(),
    "Type must be copy constructible."); ❷
  static_assert(std::is_arithmetic<T>(),
    "Type must support addition and division."); ❸
  static_assert(std::is_constructible<T, size_t>(),
    "Type must be constructible from size_t."); ❹
  --пропуск--
}
```

Вы можете заметить знакомые типажи типа для проверки того, что `T` является значением по умолчанию ❶ и может использовать конструктор копирования ❷, и вы предоставляете методы ошибок, чтобы помочь пользователям диагностировать проблемы с созданием шаблона. Используется `is_arithmetic` ❸, который вычисляется как `true`, если параметр типа поддерживает арифметические операции (+, –, /и *), и `is_constructible` ❹, который определяет, можно ли сконструировать `T` из `size_t`.

Использование `static_assert` в качестве прокси для концептов — это «костыль», но он широко используется. Используя типажи типа, можно хромать, пока понятия не будут включены в стандарт. Вы будете часто видеть `static_assert` при использовании современных сторонних библиотек; если вы пишете код для других (в том числе и для себя в будущем), подумайте об использовании `static_assert` и типажей типа.

Компиляторы и зачастую программисты не читают документацию. Выполняя требования непосредственно в коде, можно избежать устаревшей документации. В отсутствие концептов `static_assert` является хорошим временным решением.

Нетиповые параметры шаблона

Параметр шаблона, объявленный с ключевым словом `typename` (или `class`), называется *типовым параметром шаблона*, который заменяет некоторый тип, который еще не определен. В качестве альтернативы можно использовать *нетиповые параметры шаблона*, которые являются заменой для некоторого еще не определенного значения. Нетиповые параметры шаблона могут быть одним из следующих:

- интегральный тип;
- ссылочный тип l-значений;
- тип указателя (или указателя на член);
- `std::nullptr_t` (тип `nullptr`);
- `enum class`.

Использование нетипового параметра шаблона позволяет вводить значение в общий код во время компиляции. Например, можно сконструировать функцию шаблона с именем `get`, которая проверяет доступ к массиву вне границ во время компиляции, принимая индекс, к которому нужно получить доступ, как параметр шаблона нетипичного типа.

Вспомните из главы 3, что если передавать массив функции, он превращается в указатель. Вместо этого можно передать ссылку на массив с особенно непривычным синтаксисом:

```
тип-элемента(&имя-параметра)[размер-массива]
```

Например, в листинге 6.30 содержится функция `get`, которая делает первую попытку доступа к массиву с проверкой границ.

Листинг 6.30. Функция доступа к элементам массива с проверкой границ

```
#include <stdexcept>

int& get(int (&arr)[10]❶, size_t index❷) {
  if (index >= 10) throw std::out_of_range{ "Out of bounds" }; ❸
  return arr[index]; ❹
}
```

Функция `get` принимает ссылку на массив `int` размером 10 ❶ и индекс (`index`) для извлечения ❷. Если `index` выходит за пределы массива, она генерирует исключение `out_of_ range` ❸; в противном случае возвращает ссылку на соответствующий элемент ❹.

Можно улучшить листинг 6.30 тремя способами, которые доступны с помощью нетиповых параметров шаблона, обобщающими значения из `get`.

Во-первых, можно ослабить требование, чтобы `arr` ссылался на массив `int`, сделав `get` функцией шаблона, как показано в листинге 6.31.

Листинг 6.31. Рефакторинг листинга 6.30 для принятия массива универсального типа

```
#include <stdexcept>

template <typename T❶>
T&❷ get(T❸ (&arr)[10], size_t index) {
 if (index >= 10) throw std::out_of_range{ "Out of bounds" };
 return arr[index];
}
```

Как уже было показано в этой главе, функция обобщается, заменяя конкретный тип (здесь `int`) параметром шаблона ❶❷❸.

Во-вторых, можно ослабить требование, чтобы `arr` ссылался на массив размером 10, введя нестандартный параметр шаблона `Length`. В листинге 6.32 показано, как это сделать: просто объявите параметр шаблона `size_t Length` и используйте его вместо 10.

Листинг 6.32. Рефакторинг листинга 6.31 для принятия массива любого размера

```
#include <stdexcept>

template <typename T, size_t Length❶>
T& get (T(&arr)[Length❷], size_t index) {
  if (index >= Length❸) throw std::out_of_range{ "Out of bounds" };
  return arr[index];
}
```

Идея та же: вместо того чтобы заменить определенный тип (`int`), было заменено определенное целое значение (`10`) ❶❷❸. Теперь можно использовать функцию с массивами любого размера.

В-третьих, можно выполнить проверку границ во время компиляции, приняв индекс `size_t` в качестве другого нетипового параметра шаблона. Это позволяет заменить `std::out_of_range` на `static_assert`, как показано в листинге 6.33.

Листинг 6.33. Программа, использующая доступ к массиву с проверкой границ во время компиляции

```
#include <cstdio>

template <size_t Index❶, typename T, size_t Length>
T& get(T (&arr)[Length]) {
  static_assert(Index < Length, "Out-of-bounds access"); ❷
  return arr[Index❸];
}

int main() {
  int fib[]{ 1, 1, 2, 0 }; ❹
  printf("%d %d %d ", get<0>(fib), get<1>(fib), get<2>(fib)); ❺
  get<3>(fib) = get<1>(fib) + get<2>(fib); ❻
  printf("%d", get<3>(fib)); ❼
  //printf("%d", get<4>(fib)); ❽
}
--------------------------------------------------------------------------------
1 1 2 ❺3 ❼
```

Параметр индекса `size_t` был перемещен в нетиповой параметр шаблона ❶, а доступ к массиву обновлен с указанием правильного имени `Index` ❸. Поскольку `Index` теперь является постоянной времени компиляции, также `logic_error` заменяется на `static_assert`, который выводит дружественное сообщение `Out-of-bounds` (Выход за пределы массива) всякий раз при случайной попытке получить доступ к элементу за пределами массива ❷.

Листинг 6.33 также содержит пример использования `get` в `main`. Сначала был объявлен массив `int fib` размером 4 ❹. Затем были выведены первые три элемента массива с использованием `get` ❺, установлен ❻ и выведен ❼ четвертый элемент. Если раскомментировать доступ за переделы массива ❽, компилятор выдаст ошибку благодаря `static_assert`.

Вариативные шаблоны

Иногда шаблоны должны принимать неизвестное количество аргументов. Компилятор знает эти аргументы при создании шаблона, но нужно избежать написания множества разных шаблонов для каждого числа аргументов. В этом заключается смысл вариативных шаблонов. *Вариативные шаблоны* принимают переменное количество аргументов.

Шаблоны с переменными параметрами обозначаются, используя последний параметр шаблона, имеющий специальный синтаксис, а именно `typename... arguments`. Многоточие указывает, что `arguments` — это тип пакета параметров, то есть можно объявлять пакеты параметров в шаблоне. Пакет параметров — это аргумент шаблона, который принимает ноль или более аргументов функции. Эти определения могут показаться немного заумными, поэтому рассмотрим следующий пример шаблона переменной, который основан на `SimpleUniquePointer`.

Вспомните из листинга 6.14, что необработанный указатель передается в конструктор `SimpleUniquePointer`. В листинге 6.34 реализована функция `make_simple_unique`, которая обрабатывает создание базового типа.

Листинг 6.34. Реализация функции make_simple_unique для упрощения использования SimpleUniquePointer

```
template <typename T, typename... Arguments❶>
SimpleUniquePointer<T> make_simple_unique(Arguments... arguments❷) {
  return SimpleUniquePointer<T>{ new T{ arguments...❸ } };
}
```

Определяется тип пакета параметров `Arguments` ❶, который объявляет `make_simple_unique` как шаблон с переменным числом аргументов ❷. Эта функция передает аргументы конструктору параметра шаблона `T` ❸.

В результате теперь можно очень просто создавать `SimpleUniquePointers`, даже если у указанного объекта есть конструктор не по умолчанию.

ПРИМЕЧАНИЕ

Существует немного более эффективная реализация листинга 6.34. Если `arguments` является `r`-значением, можно переместить его непосредственно в конструктор `T`. stdlib содержит функцию `std::forward` в заголовке `<utility>`, которая определяет, являются ли аргументы `l`- или `r`-значениями, и выполняет копирование или перенос соответственно. См. правило 23 в «Эффективном использовании C++» Скотта Мейерса.

Продвинутое использование шаблонов

Для повседневного полиморфного программирования шаблоны — это подходящий инструмент. Оказывается, что шаблоны также используются в широком спектре расширенных настроек, особенно при реализации библиотек, высокопроизводительных программ и встроенных программ. В этом разделе описываются некоторые основные особенности ландшафта этого обширного пространства.

Специализация шаблона

Чтобы понять продвинутое использование шаблона, нужно сначала понять специализацию шаблона. Шаблоны могут на самом деле принимать больше, чем просто концепт и параметры `typename` (параметры типа). Они также могут принимать фундаментальные типы, такие как `char` (параметры значения), а также другие шаблоны. Учитывая огромную гибкость параметров шаблона, можно принять множество решений во время компиляции об их функциях. Можно иметь разные версии шаблонов в зависимости от характеристик этих параметров. Например, если параметром типа является `Ordered` вместо `Regular`, возможно, вы сможете сделать универсальную программу более эффективной. Программирование таким способом называется *специализацией шаблона*. Обратитесь к стандарту ISO [temp.spec] для получения дополнительной информации о специализации шаблона.

Связка имен

Другим важным компонентом создания шаблонов является связка имен. Связка имен помогает определить правила, когда компилятор сопоставляет именованный элемент в шаблоне с конкретной реализацией. Названный элемент может, например, быть частью определения шаблона, локального имени, глобального имени или некоторого именованного пространства имен. Если нужно написать сильно шаблонизированный код, нужно понять, как происходит эта связка. Если вы находитесь в такой ситуации, обратитесь к главе 9 «Имена в шаблонах» в разделе «*Шаблоны C++. Справочник разработчика*» Дэвида Вандевурда[1] и др., а также к [temp.res].

Функция типа

Функция типа принимает типы в качестве аргументов и возвращает тип. Типажи типа, с которыми строятся концепты, тесно связаны с функциями типов. Можно комбинировать функции типов со структурами управления компиляцией для выполнения общих вычислений, таких как программирование потока управления, во время компиляции. Обычно программирование с использованием этих методов называется *шаблонным метапрограммированием*.

---

[1] *Вандевурд Д., Джосаттис Н. М., Грегор Д.* Шаблоны C++. Справочник разработчика, 2-е изд. Пер. с англ.— СПб.: ООО «Альфа-книга», 2018.— 848 с. — *Примеч. ред.*

Метапрограммирование шаблонов

Метапрограммирование шаблонов имеет заслуженную репутацию за создание кода, который является чрезвычайно умным и абсолютно непостижимым для всех, кроме самых могущественных волшебников. К счастью, когда концепты станут частью стандарта C++, метапрограммирование шаблонов должно стать более доступным для нас, простых смертных. До тех пор действуйте осторожно. Тем, кто интересуется более подробной информацией по этой теме, нужно обратиться к книгам «*Современное проектирование на C++: Обобщенное программирование и прикладные шаблоны проектирования*» Андрея Александреску и «*Шаблоны C++. Справочник разработчика*» Дэвида Вандевурда и др.

Организация исходного кода шаблона

Каждый раз, когда создается экземпляр шаблона, компилятор должен иметь возможность генерировать весь код, необходимый для использования шаблона. Это означает, что вся информация о том, как создать экземпляр пользовательского класса или функции, должна быть доступна в той же единице трансляции, что и экземпляр шаблона. Безусловно, самый популярный способ добиться этого — реализовать шаблоны полностью внутри заголовочных файлов.

С этим подходом связаны некоторые небольшие неудобства. Время компиляции может увеличиться, потому что шаблоны с одинаковыми параметрами могут быть созданы несколько раз. Это также уменьшает способность программиста скрывать детали реализации. К счастью, преимущества общего программирования намного перевешивают эти неудобства. (Основные компиляторы, вероятно, минимизируют проблемы времени компиляции и дублирования кода.)

Есть даже несколько преимуществ использования шаблонов только для заголовков:

- другим людям очень просто использовать ваш код: это вопрос применения `#include` к некоторым заголовкам (вместо того, чтобы компилировать библиотеку, гарантируя, что полученные объектные файлы видны редактору связей, и т. д.);
- компиляторам довольно просто встроить шаблоны только для заголовков, что может привести к более быстрому выполнению кода;
- как правило, компиляторы могут лучше оптимизировать код при наличии всего исходного кода.

Полиморфизм во время выполнения и компиляции

Если нужен полиморфизм — используйте шаблоны. Но иногда нельзя использовать шаблоны, потому что вы не будете знать типы, используемые в коде, до времени выполнения. Помните, что создание экземпляра шаблона происходит только тогда, когда вы связываете параметры шаблона с типами. На этом этапе компилятор может

создать пользовательский класс самостоятельно. В некоторых ситуациях не получится выполнять такие спаривания до тех пор, пока программа не будет выполнена (или, по крайней мере, выполнение такого спаривания во время компиляции будет утомительным).

В таких случаях можно использовать динамический полиморфизм. В то время как шаблон является механизмом для достижения полиморфизма во время компиляции, механизм времени выполнения является интерфейсом.

Итоги

В этой главе вы исследовали полиморфизм в C++. Глава началась с обсуждения того, что такое полиморфизм и почему он так полезен. Вы узнали, как добиться полиморфизма во время компиляции с помощью шаблонов. Вы узнали о проверке типов с помощью концептов, а затем изучили некоторые сложные темы, такие как вариативные шаблоны и метапрограммирование шаблонов.

Упражнения

6.1. Режим ряда значений — это значение, которое появляется чаще всего. Реализуйте функцию `mode` («режим»), используя следующую сигнатуру: `int mode(const int * values, size_t length)`. Если вы столкнулись с ошибкой, такой как ввод с несколькими режимами и без значений, верните ноль.

6.2. Реализуйте `mode` как шаблонную функцию.

6.3. Модифицируйте `mode`, чтобы принимать концепт `Integer`. Убедитесь, что `mode` не может быть создана с типами с плавающей точкой, такими как `double`.

6.4. Перепишите `mean` из листинга 6.13 для принятия массива, а не указателя и аргументов длины. Используйте листинг 6.33 как руководство.

6.5. Используя пример из главы 5, сделайте `Bank` классом шаблона, который принимает параметр шаблона. Используйте этот параметр типа в качестве типа учетной записи вместо`long`. Убедитесь, что код все еще работает, используя класс `Bank<long>`.

6.6. Реализуйте класс `Account` и создайте экземпляр `Bank <Account>`. Реализуйте функции в `Account`, чтобы отслеживать баланс.

6.7. Сделайте `Account` интерфейсом. Реализуйте `CheckingAccount` («расчетный счет») и `SavingsAccount` («сберегательный счет»). Создайте программу с несколькими расчетными и сберегательными счетами. Используйте `Bank<Account>` для совершения нескольких транзакций между счетами.

Что еще почитать?

- «Шаблоны C++. Справочник разработчика», 2-е издание, Дэвид Вандевурд, Николай М. Джосаттис и Дуглас Грегор (Вильямс, 2018)
- «Эффективный и современный C++. 42 рекомендации по использованию C++11 и C++14», Скотт Мейерс (Вильямс, 2019)
- «Язык программирования C++», 4-е издание, Бьёрн Страуструп (Бином, 2011)
- «Современное проектирование на C++: Обобщенное программирование и прикладные шаблоны проектирования», Андрей Александреску (Вильямс, 2004)

7 Выражения

Здесь заключена суть созидательного гения человека: не величественные амбиции цивилизации и не разрушительное оружие, способное мгновенно обратить их в пыль, но лишь слова, которые, подобно атакующим яйцеклетку сперматозоидам, оплодотворяют новые концепции.

Дэн Симмонс, «Гиперион»

Выражения — это вычисления, которые дают результаты и побочные эффекты. Как правило, выражения содержат операнды и операторы, которые работают с ними. Многие операторы включены в основной язык, и вы увидите большинство из них в этой главе. Глава начинается с обсуждения встроенных операторов, переходит к обсуждению оператора перегрузки `new` и определяемых пользователем литералов, а затем углубляется в изучение преобразований типов. При создании собственных пользовательских типов часто нужно описывать, как эти типы преобразуются в другие типы. Вы изучите эти пользовательские преобразования, прежде чем узнаете о константных выражениях `constexpr` и повсеместно неправильно понимаемом ключевом слове `volatile`.

Операторы

Операторы, такие как операторы сложения (+) и вычисления адреса (&), работают с аргументами, называемыми операндами, такими как числовые значения или объекты. В этом разделе мы рассмотрим операторы логические, арифметические, присваивания, увеличения/уменьшения, сравнения, доступа к элементам, тернарные условные и операторы запятой.

Логические операторы

Набор выражений C++ включает в себя полный набор логических операторов. В эту категорию входят (обычно) операторы логические И (`&&`), ИЛИ (`||`) и НЕ (`!`), которые принимают конвертируемые в `bool` операнды и возвращают объект типа `bool`. Также *побитовые логические операторы* работают с целочисленными типами, такими как `bool`, `int` и `unsigned long`. Эти операторы включают И (`&`), ИЛИ (`|`), исключающее ИЛИ (`^`), дополнение (`~`), сдвиг влево (`<<`) и сдвиг вправо (`>>`). Каждый из них выполняет логическую операцию на битовом уровне и возвращает целочисленный тип, соответствующий его операндам.

В таблице 7.1 перечислены все эти логические операторы наряду с некоторыми примерами.

Таблица 7.1. Логические операторы

Оператор	Имя	Пример выражения	Результат
`x & y`	Побитовое И	`0b1100 & 0b1010`	`0b1000`
`x | y`	Побитовое ИЛИ	`0b1100 | 0b1010`	`0b1110`
`x ^ y`	Побитовое исключающее ИЛИ	`0b1100 ^ 0b1010`	`0b0110`
`~x`	Побитовое дополнение	`~0b1010`	`0b0101`
`x << y`	Побитовый сдвиг влево	`0b1010 << 2` `0b0011 << 4`	`0b101000` `0b110000`
`x >> y`	Побитовый сдвиг вправо	`0b1010 >> 2` `0b10110011 >> 4`	`0b10` `0b1011`
`x && y`	И	`true && false` `true && true`	`false` `true`
`x || y`	ИЛИ	`true || false` `false || false`	`true` `false`
`!x`	НЕ	`!true` `!false`	`false` `true`

Арифметические операторы

Дополнительные унарные и бинарные *арифметические операторы* работают как с целочисленными типами, так и с типами с плавающей точкой (также называемыми *арифметическими типами*). Вы будете использовать встроенные арифметические операторы везде, где нужно выполнять математические вычисления. Они выполняют некоторые из самых основных элементов работы независимо от того, увеличиваете ли вы индексную переменную или выполняете статистически интенсивное моделирование.

Унарные арифметические операторы

Унарный оператор плюс + и *унарный минус* принимают один арифметический операнд. Оба оператора *переводят* свои операнды в `int`. Итак, если операнд имеет тип `bool`, `char` или `shortint`, результатом выражения будет `int`.

Унарный плюс мало что делает кроме перевода; унарный минус, с другой стороны, меняет знак операнда. Например, учитывая, что `charx = 10`, `+x` приводит к `int` со значением 10, а `–x` приводит к `int` со значением –10.

Бинарные арифметические операторы

Помимо двух унарных арифметических операторов существуют пять *бинарных* арифметических операторов: *сложение* +, *вычитание* –, *умножение* *, *деление* / и *деление по модулю* %. Эти операторы принимают два операнда и выполняют указанную математическую операцию. Как и их унарные аналоги, эти бинарные операторы вызывают целочисленный перевод своих операндов. Например, сложение двух операндов `char` приведет к `int`. Также существуют правила перевода чисел с плавающей точкой:

- если один операнд `long double`, другой операнд переводится в `long double`;
- если один операнд `double`, другой операнд переводится в `double`;
- если один операнд `float`, другой операнд переводится в `float`.

Если ни одно из правил перевода чисел с плавающей запятой неприменимо, вы проверяете, является ли какой-либо из аргументов знаковым. Если это так, оба операнда становятся знаковыми. Наконец, как и в правилах перевода для типов с плавающей точкой, размер наибольшего операнда используется для перевода другого операнда:

- если один операнд `long long`, другой операнд переводится в `long long`;
- если один операнд `long`, другой операнд переводится в `long`;
- если один операнд `int`, другой операнд переводится в `int`.

Хотя эти правила не слишком сложны для запоминания, я рекомендую проверять работу, опираясь на вывод типа с помощью `auto`. Просто присвойте результат выражения `auto` выведенной переменной и проверьте выведенный тип.

Не путайте приведение и перевод. Приведение — это когда у вас есть объект одного типа и нужно преобразовать его в другой тип. Перевод — это набор правил для интерпретации литералов. Например, если есть платформа с 2-байтовым коротким значением и вы выполнили преобразование со знаком для `unsigned short` со значением 40000, результатом будут переполнение целого числа и неопределенное поведение. Это полностью отличается от обработки правил перевода для литерала 40000. Если необходим знаковый тип, тип литерала будет знаковым `int`, потому что знаковый `short` недостаточно велик, чтобы вместить такое значение.

ПРИМЕЧАНИЕ

Вы можете использовать IDE или даже `typeid` в RTTI, чтобы вывести тип в консоль.

Таблица 7.2 обобщает арифметические операторы.

Таблица 7.2. Арифметические операторы

Оператор	Имя	Примеры	Результат
`+x`	Унарный плюс	`+10`	`10`
`-x`	Унарный минус	`-10`	`-10`
`x + y`	Бинарное сложение	`1 + 2`	`3`
`x - y`	Бинарное вычитание	`1 – 2`	`-1`
`x * y`	Бинарное умножение	`10 * 20`	`200`
`x/y`	Бинарное деление	`300/15`	`20`
`x % y`	Бинарное деление по модулю	`42 % 5`	`2`

Многие из бинарных операторов в табл. 7.1 и 7.2 также имеют следствия в качестве *операторов присваивания*.

Операторы присваивания

Оператор присваивания выполняет данную операцию, а затем присваивает результат первому операнду. Например, *присваивание сложения* `x += y` вычисляет значение `x + y` и присваивает `x` значение результата. Можно получить аналогичные результаты с помощью выражения `x = x + y`, но *оператор присваивания* более синтаксически компактен и, по крайней мере, эффективен во время выполнения. Таблица 7.3 обобщает все доступные операторы присваивания.

Таблица 7.3. Операторы присваивания

Оператор	Имя	Примеры	Результат (значение x)
`x = y`	Простое присваивание	`x = 10`	`10`
`x += y`	Присваивание сложения	`x += 10`	`15`
`x -= y`	Присваивание вычитания	`x -= 10`	`-5`
`x *= y`	Присваивание умножения	`x *= 10`	`50`
`x /= y`	Присваивание деления	`x /= 2`	`2`
`x %= y`	Присваивание деления по модулю	`x %= 2`	`1`
`x &= y`	Присваивание побитового И	`x &= 0b1100`	`0b0100`
`x |= y`	Присваивание побитового ИЛИ	`x |= 0b1100`	`0b1101`
`x ^= y`	Присваивание побитового исключающего ИЛИ	`x ^= 0b1100`	`0b1001`
`x <<= y`	Присваивание побитового сдвига влево	`x <<= 2`	`0b10100`
`x >>= y`	Присваивание побитового сдвига вправо	`x >>= 2`	`0b0001`

ПРИМЕЧАНИЕ

Правила перевода не применяются при использовании операторов присваивания; тип назначенного операнда не изменится. Например, если задано `int x = 5`, тип `x` после `x /= 2.0f` будет по-прежнему `int`.

Операторы увеличения и уменьшения

Существует четыре (унарных) оператора *увеличения/уменьшения* (инкремента/декремента), как показано в табл. 7.4.

Таблица 7.4. Операторы увеличения и уменьшения (значения приведены для `x=5`)

Оператор	Имя	Значение x после вычисления	Значение выражения
`++x`	Префиксное увеличение	6	6
`x++`	Постфиксное увеличение	6	5
`--x`	Префиксное уменьшение	4	4
`x--`	Постфиксное уменьшение	4	5

Как показано в табл. 7.4, операторы увеличения увеличивают значение своего операнда на 1, тогда как операторы уменьшения уменьшают на 1. Значение, возвращаемое оператором, зависит от того, является ли он префиксным или постфиксным. Префиксный оператор вернет значение операнда после модификации, тогда как постфиксный оператор вернет значение до модификации.

Операторы сравнения

Шесть операторов сравнения сравнивают данные операнды и вычисляют их как `bool`, как показано в табл. 7.5. Для арифметических операндов происходят те же преобразования типов (переводы), что и для арифметических операторов. Операторы сравнения также работают с указателями и примерно так, как ожидается.

ПРИМЕЧАНИЕ

Есть несколько нюансов в сравнении указателей. Заинтересованным читателям стоит обратиться к [expr.rel].

Операторы доступа к членам

Операторы доступа к членам используются для взаимодействия с указателями, массивами и многими классами, которые вы встретите во второй части. Шесть таких операторов включают в себя *индексирование* [], *разыменование* *, *вычисление*

адреса &, *доступа к членам* и *доступа к членам через указатель* >. Вы познакомились с этими операторами в главе 3, но в этом разделе приводится краткая сводка по ним.

Таблица 7.5. Операторы сравнения

Оператор	Имя	Примеры (все вычисляются как `true`)
`x == y`	Оператор равенства	`100 == 100`
`x != y`	Оператор неравенства	`100 != 101`
`x < y`	Оператор «меньше»	`10 < 20`
`x > y`	Оператор «больше»	`-10 > -20`
`x <= y`	Оператор «меньше или равно»	`10 <= 10`
`x >= y`	Оператор «больше или равно»	`20 >= 10`

ПРИМЕЧАНИЕ

Существуют также операторы указателя на доступ к членам `.*` и указателя на доступ к членам через указатель `->*`, но они встречаются редко. Обратитесь к [expr.mptr.oper].

Оператор индексирования `x[y]` обеспечивает доступ к `y`-му элементу массива, на который указывает `x`, тогда как оператор разыменования `*x` обеспечивает доступ к элементу, на который указывает `x`. Можно создать указатель на элемент `x` с помощью оператора вычисления адреса `&x`. По сути, это обратная операция для оператора разыменования. Для элементов `x` с членом `y` используется оператор доступа к членам `x.y`. Также можно получить доступ к членам указанного объекта через указатель; учитывая указатель `x`, используйте оператор доступа к членам через указатель `x->y` для доступа к объекту, на который указывает `x`.

Тернарные условные операторы

Тернарный условный оператор `x? y: z` — кусочек синтаксического сахара, который принимают три операнда (отсюда «тернарный»). Он оценивает первый операнд `x` как логическое выражение и возвращает второй операнд `y` или третий операнд `z` в зависимости от того, является ли логическое значение `true` или `false` (соответственно). Рассмотрим следующую пошаговую функцию, которая возвращает 1, если входной параметр положительный; в противном случае возвращается ноль:

```
int step(int input) {
  return input > 0 ? 1 : 0;
}
```

Используя эквивалентный оператор `if-then`, можно реализовать `step` следующим образом:

```
int step(int input) {
  if (input > 0) {
    return 1;
  } else {
    return 0;
  }
}
```

Эти два подхода эквивалентны во время выполнения, но тернарный условный оператор требует меньше места и приводит к более чистому коду. Стоит щедро его использовать.

ПРИМЕЧАНИЕ

Условный тернарный оператор имеет модное имя: оператор Элвиса. Если вы повернете книгу на 90 градусов по часовой стрелке и присмотритесь, то поймете, почему: `?:`.

Оператор запятой

Оператор запятой, с другой стороны, обычно не продвигает более чистый код. Это позволяет нескольким выражениям, разделенным запятыми, вычисляться в большем выражении. Выражения оцениваются слева направо, а крайнее правое выражение — это возвращаемое значение, как показано в листинге 7.1.

Листинг 7.1. Функция confusing, использующая оператор запятой

```
#include <cstdio>

int confusing(int &x) {
  return x = 9, x++, x / 2;
}

int main() {
  int x{}; ❶
  auto y = confusing(x); ❷
  printf("x: %d\ny: %d", x, y);
}
--------------------------------------------------------------------------------
x: 10
y: 5
```

После вызова `confusing` `x` равен `10` ❶, а `y` равен `5` ❷.

ПРИМЕЧАНИЕ

Оператор запятой допускает особый вид программирования — ориентированное на выражения. Избегайте оператора запятой; его использование чрезвычайно редко и может вызвать путаницу.

Перегрузка операторов

Для каждого основного типа будет доступна некоторая часть операторов, описанных в этом разделе. Для пользовательских типов можно указать пользовательское поведение этих операторов, используя *перегрузку операторов*. Чтобы указать поведение оператора в пользовательском классе, просто назовите метод словом `operator`, за которым сразу следует оператор; убедитесь, что возвращаемые типы и параметры соответствуют типам операндов, с которыми предполагается иметь дело.

Листинг 7.2 определяет `CheckedInteger`.

Листинг 7.2. Класс CheckedInteger, который обнаруживает переполнение во время выполнения

```
#include <stdexcept>

struct CheckedInteger {
  CheckedInteger(unsigned int value) : value{ value } ❶ { }

  CheckedInteger operator+(unsigned int other) const { ❷
    CheckedInteger result{ value + other }; ❸
    if (result.value < value) throw std:: overflow_error{ "Overflow!" }; ❹
    return result;
  }

  const unsigned int value; ❺
};
```

В этом классе определен конструктор, который принимает один `unsigned int`. Этот аргумент используется ❶ для инициализации публичного поля `value` ❺. Поскольку `value` помечено как `const`, `CheckedInteger` остается *неизменным* — после создания невозможно изменить состояние `CheckedInteger`. Интересующий нас метод здесь — `operator+` ❷, который позволяет добавить обычное беззнаковое целое к `CheckedInteger` для создания нового `CheckedInteger` с правильным `value`. Возвращаемое значение `operator+` создается в ❸. Всякий раз, когда добавление приводит к переполнению `unsigned int`, результат будет меньше, чем исходные значения. Это условие проверяется в ❹. Если переполнение обнаружено, выбрасывается исключение.

В главе 6 описаны `type_traits`, которые позволяют определять особенности типов во время компиляции. Связанное семейство поддержки типов доступно в заголовке `<limits>`, что позволяет запрашивать различные свойства арифметических типов.

В пределах `<limits>` класс шаблона `numeric_limits` предоставляет ряд константчленов, которые предоставляют информацию о параметре шаблона. Одним из таких примеров является метод `max()`, который возвращает наибольшее конечное значение заданного типа. Можно использовать этот метод, чтобы оценить класс `CheckedInteger`. Листинг 7.3 показывает поведение `CheckedInteger`.

Листинг 7.3. Программа с CheckedInteger

```
#include <limits>
#include <cstdio>
#include <stdexcept>

struct CheckedInteger {
  --пропуск--
};

int main() {
  CheckedInteger a{ 100 }; ❶
  auto b = a + 200; ❷
  printf("a + 200 = %u\n", b.value);
  try {
    auto c = a + std::numeric_limits<unsigned int>::max(); ❸
  } catch(const std::overflow_error& e) {
    printf("(a + max) Exception: %s\n", e.what());
  }
}
--------------------------------------------------------------------------------
a + 200 = 300
(a + max) Exception: Overflow!
```

После создания `CheckedInteger` ❶ можно добавить его к `unsigned int` ❷. Поскольку результирующее значение 300 гарантированно помещается в `unsigned int`, этот оператор выполняется без исключения. Затем добавляется тот же `CheckedInteger`, а к максимальному значению — `unsigned int` через `numeric_limits` ❸. Это вызывает переполнение, которое обнаруживается перегрузкой `operator+` и приводит к выбрасыванию `overflow_error`.

Оператор перегрузки new

Вспомните (глава 4), что можно выделить объекты с динамической длительностью хранения, используя оператор `new`. По умолчанию оператор `new` выделяет память в *свободном хранилище*, чтобы освободить место для динамических объектов. Свободное хранилище, также известное как *куча*, является местом хранения, определяемым реализацией. В настольных операционных системах ядро обычно управляет свободным хранилищем (см. `HeapAlloc` в Windows и `malloc` в Linux и macOS).

Доступность свободного хранилища

В некоторых средах, таких как ядро Windows или встроенные системы, свободное хранилище по умолчанию недоступно. В других разработках, таких как разработка игр или высокочастотная торговля, свободное хранилище привносит сильную задержку, потому что управление им делегировано операционной системе.

Можно попытаться полностью избежать использования свободного хранилища, но это сильно ограничивает разработку. Одно из основных ограничений, которое это может ввести, состоит в том, чтобы исключить использование контейнеров stdlib,

и после прочтения части 2 вы согласитесь, что это серьезная потеря. Вместо того чтобы соглашаться с этими серьезными ограничениями, можно перегрузить операции свободного хранилища и взять под контроль распределение. Достигается это путем перегрузки оператора `new`.

Заголовок <new>

В средах, которые поддерживают операции над свободным хранилищем, заголовок `<new>` содержит следующие четыре оператора:

- `void *operator new (size_t);`
- `void operator delete(void*);`
- `void* operator new [] (size_t);`
- `void operator void delete [] (void*).`

Обратите внимание, что возвращаемый тип оператора `new` — `void*`. Операторы свободного хранилища работают с сырой неинициализированной памятью.

Можно предоставить собственные версии этих четырех операторов. Все, что нужно сделать, — это определить их один раз в программе. Компилятор будет использовать пользовательские версии, а не значения по умолчанию.

Управление свободным хранилищем — удивительно сложная задача. Одна из главных проблем — *фрагментация памяти*. Со временем большое количество выделений и выпусков памяти может оставить свободные блоки памяти, предназначенные для свободного хранилища и разбросанные по всему пространству. Можно оказаться в ситуации, когда имеется много свободной памяти, но она разбросана по выделенной памяти. Когда это произойдет, большие запросы в памяти потерпят неудачу, даже если технически свободной памяти достаточно для предоставления запрашивающей стороне. На рис. 7.1 приведена такая ситуация. Памяти достаточно для желаемого выделения, но доступная память не является непрерывной.

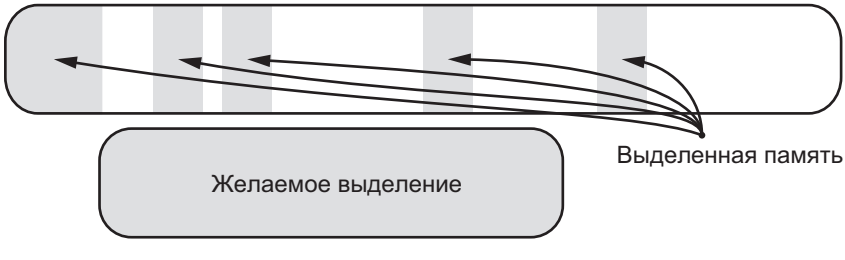


Рис. 7.1. Проблема фрагментации памяти

Сегменты

Один из подходов заключается в разделении выделенной памяти на так называемые *сегменты* фиксированного размера. При запросе памяти среда выделяет

целый сегмент, даже если вы не запрашивали всю память. Например, Windows предоставляет две функции для выделения динамической памяти: `VirtualAllocEx` и `HeapAlloc`.

Функция `VirtualAllocEx` является низкоуровневой, что позволяет предоставлять множество опций, таких как процесс выделения памяти, предпочтительный адрес памяти, запрошенный размер и разрешения, например должна ли память быть читаемой, записываемой и исполняемой. Эта функция никогда не выделит менее 4096 байт (так называемая *страница*).

С другой стороны, `HeapAlloc` — это высокоуровневая функция, которая раздает меньше страницы памяти, когда может; в противном случае она вызовет `VirtualAllocEx` от вашего имени. Это работает так, по крайней мере, с компилятором Visual Studio — `new` вызовет `HeapAlloc` по умолчанию.

Такое расположение предотвращает фрагментацию памяти в обмен на некоторые издержки, связанные с округлением выделения до размера сегмента. Современные операционные системы, такие как Windows, имеют довольно сложные схемы для выделения памяти разных размеров. Вы не увидите ничего подобного, пока не захотите взять контроль в свои руки.

Контроль над свободным хранилищем

Листинг 7.4 демонстрирует реализацию очень простых классов `Bucket` и `Heap`. Они облегчат управление динамическим распределением памяти.

Листинг 7.4. Классы Heap и Bucket

```
#include <cstddef>
#include <new>

struct Bucket { ❶
  const static size_t data_size{ 4096 };
  std::byte data[data_size];
};

struct Heap {
  void* allocate(size_t bytes) { ❷
    if (bytes > Bucket::data_size) throw std::bad_alloc{};
    for (size_t i{}; i < n_heap_buckets; i++) {
      if (!bucket_used[i]) {
        bucket_used[i] = true;
        return buckets[i].data;
      }
    }
    throw std::bad_alloc{};
   }

   void free(void* p) { ❸
     for (size_t i{}; i < n_heap_buckets; i++) {
       if (buckets[i].data == p) {
```

```
          bucket_used[i] = false;
          return;
        }
      }
    }
    static const size_t n_heap_buckets{ 10 };
    Bucket buckets[n_heap_buckets]{}; ❹
    bool bucket_used[n_heap_buckets]{}; ❺
};
```

Класс `Bucket` ❶ отвечает за использование места в памяти. Как дань уважения диспетчеру кучи Windows размер сегмента жестко задан в 4096. Вся логика управления переходит в класс `Heap`.

Два важных учетных элемента находятся в `Heap`: `buckets` ❹ и `bucket_used` ❺. Член `buckets` содержит все `Buckets`, аккуратно упакованные в непрерывную строку. Член `bucket_used` — это сравнительно крошечный массив, содержащий объекты типа `bool`, который отслеживает, был ли выдан `Bucket` в `buckets` с тем же индексом. Оба члена инициализируются нулем.

Класс `Heap` имеет два метода: `allocate` ❷ и `free` ❸. Метод `allocate` сначала проверяет, больше ли запрошенное количество байтов, чем размер сегмента. Если это так, он генерирует исключение `std::bad_alloc`. После того как проверка размера пройдена, `Heap` перебирает сегменты, ища тот, который не помечен как `true` в `bucket_used`. Если он находит его, он возвращает указатель на член `data` для соответствующего `Bucket`. Если он не может найти неиспользуемый `Bucket`, он генерирует исключение `std::bad_alloc`. Метод `free` принимает `void*` и перебирает все сегменты, ища соответствующий указатель на член `data`. Если он находит его, он устанавливает `bucket_used` для соответствующего сегмента в `false` и возвращает результат.

Использование кучи

Один из способов выделить `Heap` — объявить его в области пространства имен, чтобы у него была статическая длительность хранения. Поскольку время жизни начинается при запуске программы, можно использовать кучу внутри перегрузок `operator new` и `operator delete`, как показано в листинге 7.5.

Листинг 7.5. Переопределение операторов new и delete для использования класса Heap из листинга 7.4

```
Heap heap; ❶

void* operator new(size_t n_bytes) {
  return heap.allocate(n_bytes); ❷
}

void operator delete(void* p) {
  return heap.free(p); ❸
}
```

Листинг 7.5 объявляет `Heap` ❶ и использует его внутри перегрузки оператора `new` ❷ и оператора `delete` ❸. Теперь при использовании `new` и `delete` динамическое управление памятью будет использовать `heap` вместо свободного хранилища по умолчанию, предлагаемого средой. Листинг 7.6 проливает свет на перегруженное динамическое управление памятью.

Листинг 7.6. Программа, где показано использование Heap для управления динамическим распределением

```
#include <cstdio>
--пропуск--
int main() {
  printf("Buckets: %p\n", heap.buckets); ❶
  auto breakfast = new unsigned int{ 0xC0FFEE };
  auto dinner = new unsigned int { 0xDEADBEEF };
  printf("Breakfast: %p 0x%x\n", breakfast, *breakfast); ❷
  printf("Dinner: %p 0x%x\n", dinner, *dinner); ❸
  delete breakfast;
  delete dinner;
  try {
    while (true) {
      new char;
      printf("Allocated a char.\n"); ❹
    }
  } catch (const std::bad_alloc&) {
    printf("std::bad_alloc caught.\n"); ❺
  }
}
--------------------------------------------------------------------------------
Buckets:   00007FF792EE3320 ❶
Breakfast: 00007FF792EE3320 0xc0ffee ❷
Dinner:    00007FF792EE4320 0xdeadbeef ❸
Allocated a char. ❹
Allocated a char.
Allocated a char.
Allocated a char.
Allocated a char.
Allocated a char.
Allocated a char.
Allocated a char.
Allocated a char.
Allocated a char.
Allocated a char.
std::bad_alloc caught. ❺
```

Код выводит адрес памяти первого элемента `buckets` в `heap` ❶. Это место в памяти, выделенное для первого нового вызова. Вы проверяете, что это так, печатая адрес памяти и значение, на которое указывает `breakfast` ❷. Обратите внимание, что адрес памяти совпадает с адресом памяти первого `Bucket` в `heap`. То же самое делается и для памяти, на которую указывает `dinner` ❸. Обратите внимание, что адрес памяти на 0x1000 больше, чем адрес `breakfast`. Это в точности совпадает с 4096-байтовой длиной `Bucket`, как определено в `conststatic` члене `Bucket::data_size`.

После вывода ❷❸ `breakfast` и `dinner` удаляются. Затем объекты `char` безрассудно распределяются до тех пор, пока не будет выброшено `std::bad_alloc`, когда `heap` не хватит памяти. Каждый раз при распределении выводится `Allocated the char.`, начиная с ❹. У вас есть 10 строк кода перед тем, как увидеть исключение `std::bad_alloc` ❺. Обратите внимание, что это именно то количество блоков, которое было установлено в `Heap::n_heap_buckets`. Это означает, что на каждый выделенный символ уходит 4096 байт памяти!

Операторы размещения

Иногда нет необходимости отменять *все* свободные распределения памяти. В таких ситуациях можно использовать операторы размещения, которые выполняют соответствующую инициализацию предварительно выделенной памяти:

- `void* operator new(size_t, void*);`
- `void operator delete(size_t, void*);`
- `void* operator new[](void*, void*);`
- `void operator delete[](void*, void*).`

Используя операторы размещения, можно вручную создавать объекты в свободной памяти. Преимущество этого состоит в том, что можно вручную управлять временем жизни объекта. Однако нельзя использовать `delete` для освобождения полученных динамических объектов. Необходимо вызывать деструктор объекта напрямую (и ровно один раз!), как показано в листинге 7.7.

Листинг 7.7. Использование оператора размещения new для инициализации динамических объектов

```
#include <cstdio>
#include <cstddef>
#include <new>

struct Point {
  Point() : x{}, y{}, z{} {
    printf("Point at %p constructed.\n", this); ❶
  }
  ~Point() {
    printf("Point at %p destructed.\n", this); ❷
  }
  double x, y, z;
};

int main() {
  const auto point_size = sizeof(Point);
  std::byte data[3 * point_size];
  printf("Data starts at %p.\n", data); ❸
  auto point1 = new(&data[0 * point_size]) Point{}; ❹
  auto point2 = new(&data[1 * point_size]) Point{}; ❺
  auto point3 = new(&data[2 * point_size]) Point{}; ❻
  point1->~Point(); ❼
```

```
  point2->~Point(); ❽
  point3->~Point(); ❾
}
--------------------------------------------------------------------------------
Data starts at 0000004D290FF8E8. ❸
Point at 0000004D290FF8E8 constructed. ❹
Point at 0000004D290FF900 constructed. ❺
Point at 0000004D290FF918 constructed. ❻
Point at 0000004D290FF8E8 destructed. ❼
Point at 0000004D290FF900 destructed. ❽
Point at 0000004D290FF918 destructed. ❾
```

Конструктор ❶ печатает сообщение, указывающее, что `Point` по определенному адресу был создан, а деструктор ❷ выводит соответствующее сообщение, указывающее, что `Point` уничтожается. Выводится адрес данных, который является первым адресом, где `new` будет инициализировать `Point` ❸.

Обратите внимание, что каждое новое размещение выделяет `Point` в памяти, занятой массивом `data` ❹❺❻. Нужно вызывать каждый деструктор отдельно ❼❽❾.

Приоритет операторов и ассоциативность

Когда в выражении появляется более одного оператора, *приоритет оператора* и его *ассоциативность* определяют способ анализа выражения. Операторы с более высоким приоритетом более тесно связаны со своими аргументами, чем операторы с более низким приоритетом. Если два оператора имеют одинаковый приоритет, их ассоциативность разрывает связь, чтобы решить, как связаны аргументы. Ассоциативность может быть определена *слева направо* или *справа налево*.

Таблица 7.6 содержит все операторы C++, отсортированные по приоритету, с пометками об их ассоциативности. Каждая строка содержит один или несколько операторов с одинаковым приоритетом, а также описание и ассоциативность. Чем выше расположена строка, тем выше приоритет.

Таблица 7.6. Приоритетность и ассоциативность операторов

Оператор	Описание	Ассоциативность
`a::b`	Разрешение контекста	Слева направо
`a++`	Постфиксное увеличение	Слева направо
`a--`	Постфиксное уменьшение	
`fn()`	Вызов функции	
`a[b]`	Индексирование	
`a->b`	Доступ к членам через указатель	
`a.b`	Доступ к членам	

Оператор	Описание	Ассоциативность
`Type(a)`	Функциональное приведение	
`Type{ a }`	Функциональное приведение	
`++a`	Префиксное увеличение	Справа налево
`--a`	Префиксное уменьшение	
`+a`	Унарный плюс	
`=a`	Унарный минус	
`!a`	Логическое НЕ	
`~a`	Побитовое исключающее ИЛИ	
`(Type)a`	Приведение в стиле C	
`*a`	Разыменование	
`&a`	Вычисление адреса	
`sizeof(Type)`	Вычисление размера	
`new Type`	Динамическое выделение памяти	
`new Type[]`	Динамическое выделение памяти (массив)	
`delete a`	Динамическое освобождение памяти	
`delete[] a`	Динамическое освобождение памяти (массив)	
`.*`	Указатель на доступ к членам через указатель	Слева направо
`->*`	Указатель на доступ к членам	
`a * b`	Умножение	Слева направо
`a/b`	Деление	
`a % b`	Деление по модулю	
`a + b`	Сложение	Слева направо
`a – b`	Вычитание	
`a << b`	Побитовый сдвиг влево	Слева направо
`a >> b`	Побитовый сдвиг вправо	
`a < b`	Меньше	Слева направо
`a > b`	Больше	
`a <= b`	Меньше или равно	
`a >= b`	Больше или равно	
`a == b`	Равенство	Слева направо
`a != b`	Неравенство	
`a & b`	Побитовое И	Слева направо

Продолжение ⇩

Таблица 7.6 *(продолжение)*

Оператор	Описание	Ассоциативность
`a ^ b`	Побитовое исключающее ИЛИ	Слева направо
`a | b`	Побитовое ИЛИ	Слева направо
`a && b`	Логическое И	Слева направо
`a || b`	Логическое ИЛИ	Слева направо
`a ? b : c`	Тернарный оператор	
`throw a`	Выбрасывание исключения	
`a = b`	Присваивание	
`a += b`	Присваивание суммы	
`a -= b`	Присваивание разности	
`a *= b`	Присваивание произведения	
`a /= b`	Присваивание частного	
`a %= b`	Присваивание остатка от деления	
`a <<= b`	Присваивание побитового сдвига влево	
`a >>= b`	Присваивание побитового сдвига вправо	
`a &= b`	Присваивание побитового ИЛИ	
`a ^= b`	Присваивание побитового исключающего ИЛИ	
`a |= b`	Присваивание побитового ИЛИ	
`a, b`	Запятая	Слева направо

ПРИМЕЧАНИЕ

Оператор разрешения контекста еще не встречался в этой книге (он впервые появится в главе 8), но табл. 7.6 включает его для полноты.

Поскольку в C++ есть много операторов, может быть сложно отслеживать правила приоритетов операторов и ассоциативности. Ради психического здоровья тех, кто читает код, постарайтесь сделать выражения как можно более ясными.

Рассмотрим следующее выражение:

```
*a++ + b * c
```

Поскольку постфиксное увеличение имеет более высокий приоритет, чем оператор разыменования *, оно сначала связывается с аргументом `a`, то есть результат `a++` является аргументом оператора разыменования. Умножение * имеет более высокий приоритет, чем сложение +, поэтому оператор умножения * связывается с `b` и `c`, а оператор сложения + связывается с результатами `*a++` и `b * c`.

Можно поменять приоритет в выражении, добавив скобки, которые имеют более высокий приоритет, чем любой оператор. Например, можно переписать предыдущее выражение, используя скобки:

```
(*(a++)) + (b * c)
```

Как правило, добавляйте скобки везде, где читатель может запутаться в приоритетах операторов. Если результат немного уродлив (как в этом примере), выражение, вероятно, слишком сложное; рассмотрите возможность разбиения его на несколько утверждений.

Порядок вычисления

Порядок вычисления определяет последовательность выполнения операторов в выражении. Распространенным заблуждением является то, что приоритет и порядок вычисления эквивалентны, но это не так. *Приоритет* — это концепция времени компиляции, которая определяет, как операторы связываются с операндами. *Порядок вычисления* — это концепция времени выполнения, которая управляет планированием выполнения оператора.

В общем, C++ не имеет четко определенного порядка выполнения для операндов. Хотя операторы связываются с операндами четко определенным способом, описанным в предыдущих разделах, эти операнды вычисляются в неопределенном порядке. Компилятор может упорядочить вычисление операнда по своему усмотрению.

Можно было бы подумать, что круглые скобки в следующем порядке вычисления определяют порядок вычисления выражений для функций `stop`, `drop` и `roll` или что некоторая ассоциативность слева направо имеет некоторый эффект времени выполнения:

```
(stop() + drop()) + roll()
```

Это не так. Функция `roll` может выполняться до, после или между вычислениями `stop` и `drop`. Если требуется, чтобы операции выполнялись в определенном порядке, просто поместите их в отдельные операторы в нужной последовательности, как показано здесь:

```
auto result = stop();
result = result + drop();
result = result + roll();
```

Если быть неосторожным, можно даже получить неопределенное поведение. Рассмотрим следующее выражение:

```
b = ++a + a;
```

Поскольку порядок выражений `++a` и `a` не указан и значение `++a + a` зависит от того, какое выражение вычисляется первым, значение `b` не может быть точно определено.

В некоторых особых ситуациях порядок выполнения определяется языком. Чаще всего встречаются следующие сценарии:

- встроенный логический оператор И `a && b` и встроенный логический оператор ИЛИ `a || b` гарантируют, что `a` выполняется до `b`;
- тернарный оператор `a? b: c` гарантирует, что `a` выполняется перед `b` и `c`;
- оператор запятой `a, b` гарантирует, что `a` выполняется до `b`;
- параметры конструктора в выражении `new` вычисляются перед вызовом функции распределителя.

Вам может быть интересно, почему C++ не применяет порядок выполнения, скажем, слева направо, чтобы избежать путаницы. Ответ прост: не ограничивая произвольно порядок выполнения, язык позволяет авторам компилятора находить удобные возможности оптимизации.

ПРИМЕЧАНИЕ

Для получения дополнительной информации о порядке выполнения см. [expr].

Пользовательские литералы

Глава 2 рассказывает, как объявлять литералы, постоянные значения, которые используются непосредственно в программах. Это помогает компилятору превращать встроенные значения в нужные типы. Каждый фундаментальный тип имеет свой синтаксис для литералов. Например, литерал `char` объявляется в одинарных кавычках, таких как `'J'`, тогда как `wchar_t` объявляется с префиксом `L`, таким как `L'J '`. Можно указать точность чисел с плавающей точкой, используя суффикс `F` или `L`.

Для удобства вы также можете создавать свои собственные *пользовательские литералы*. Как и в случае со встроенными литералами, они предоставляют некоторую синтаксическую поддержку для предоставления информации о типе компилятору. Хотя объявлять пользовательские литералы требуется редко, стоит упомянуть о них, потому что их можно встретить в библиотеках. Заголовок `<chrono>` в stdlib широко использует литералы, чтобы дать программистам понятный синтаксис для использования типов времени — например, `700ms` означает 700 миллисекунд. Поскольку пользовательские литералы встречаются довольно редко, я не буду здесь более подробно их описывать.

ПРИМЕЧАНИЕ

Для получения дополнительной информации см. раздел 19.2.6 4-го издания «Языка программирования C++» Бьёрна Страуструпа.

Преобразования типов

Преобразование типов выполняется, когда у вас есть один тип, но его нужно преобразовать в другой тип. В зависимости от ситуации преобразования типов могут быть явными или неявными. В этом разделе рассматриваются оба вида преобразований, при этом рассматриваются переводы, преобразования чисел с плавающей точкой в целые, преобразования целых чисел в целые и преобразования чисел с плавающей точкой в числа с плавающей точкой.

Преобразования типов довольно распространены. Например, может потребоваться вычислить среднее значение некоторых целых чисел с учетом количества и суммы. Поскольку счет и сумма хранятся в переменных целочисленного типа (и не нужно обрезать дробные значения), может понадобиться вычислить среднее значение как число с плавающей запятой. Для этого необходимо использовать преобразование типов.

Неявные преобразования типов

Неявные преобразование типов могут происходить везде, где требуется определенный тип, но предоставлен другой тип. Эти преобразования происходят в нескольких разных контекстах.

В разделе «Арифметические операторы» на с. 247 изложены так называемые *правила перевода*. На самом деле это форма неявного преобразования. Всякий раз, когда происходит арифметическая операция, более короткие целочисленные типы повышаются до типов `int`. Интегральные типы также могут быть преобразованы в типы с плавающей точкой во время арифметической операции. Все это происходит в фоновом режиме. В результате в большинстве ситуаций система типов просто скрывается из вида и можно сосредоточиться на логике программирования.

К сожалению, в некоторых ситуациях C++ немного переусердствует в тихом преобразовании типов. Рассмотрим следующее неявное преобразование `double` в `uint_8`:

```
#include <cstdint>

int main() {
  auto x = 2.7182818284590452353602874713527L;
  uint8_t y = x; // Скрытая обрезка
}
```

Нужно надеяться, что компилятор выдаст здесь предупреждение, но технически это допустимый вариант в C++. Поскольку это преобразование теряет информацию, это называется сужающим преобразованием, которое можно предотвратить с помощью фигурной инициализации {}:

```
#include <cstdint>

int main() {
  auto x = 2.7182818284590452353602874713527L;
  uint8_t y{ x }; // Бах!
}
```

Напомним, что фигурная инициализация не позволяет сужения преобразований типов. Технически инициализация со скобками является явным преобразованием, поэтому мы обсудим ее в разделе «Явное преобразование типов», с. 268.

Преобразование чисел с плавающей точкой в целые

Типы с плавающей точкой и целые могут мирно сосуществовать в арифметических выражениях. Причина заключается в неявном преобразовании типов: когда компилятор встречает смешанные типы, он выполняет необходимые переводы, поэтому арифметика продолжается, как и ожидалось.

Преобразование целых чисел в целые

Целые числа могут быть преобразованы в другие целочисленные типы. Если тип назначения `signed`, все хорошо, пока значение может быть представлено. Если это невозможно, поведение определяется реализацией. Если тип назначения `unsigned`, результат будет с таким количеством битов, которое может поместиться в тип. Другими словами, старшие биты теряются.

Рассмотрим пример из листинга 7.8, который демонстрирует, как можно получить неопределенное поведение в результате преобразования со знаком.

Листинг 7.8. Неопределенное поведение в результате преобразования знакового числа

```
#include <cstdint>
#include <cstdio>

int main() {
  // 0b111111111 = 511
  uint8_t x = 0b111111111; ❶// 255
  int8_t y =  0b111111111; ❷// Определяется реализацией.
  printf("x: %u\ny: %d", x, y);
}
--------------------------------------------------------------------------------
x: 255 ❶
y: -1 ❷
```

В листинге 7.8 неявно приводится целое число, которое слишком велико для того, чтобы вписать 8-битное целое число (511 или 9-битное) в `x` и `y`, которые соответственно являются `unsigned` и `signed`. Значение `x` гарантированно равно `255` ❶, тогда как значение `y` зависит от реализации. На компьютере с Windows 10 x64 `y` равно `-1` ❷. Присваивание `x` и `y` включает в себя сужающие преобразования, которых можно избежать, используя синтаксис фигурной инициализации.

Преобразования чисел с плавающей точкой в числа с плавающей точкой

Числа с плавающей точкой могут быть неявно приведены к и от других чисел с плавающей точкой. Пока целевое значение может соответствовать исходному значению, все в порядке. Когда это невозможно, возникает неопределенное поведение. Опять же

фигурная инициализация может предотвратить потенциально опасные преобразования. Рассмотрите пример в листинге 7.9, который демонстрирует неопределенное поведение, являющееся результатом сужения преобразования.

Листинг 7.9. Неопределенное поведение в результате сужения преобразования

```
#include <limits>
#include <cstdio>

int main() {
  double x = std::numeric_limits<float>::max(); ❶
  long double y = std::numeric_limits<double>::max(); ❷
  float z = std::numeric_limits<long double>::max(); ❸ // Неопределенное поведение
  printf("x: %g\ny: %Lg\nz: %g", x, y, z);
}
--------------------------------------------------------------------------------
x: 3.40282e+38
y: 1.79769e+308
z: inf
```

Существуют полностью безопасные неявные преобразования из `float` в `double` ❶ и из `double` в `long double` ❷ соответственно. К сожалению, присвоение максимального значения `long double` в `float` приводит к неопределенному поведению ❸.

Преобразование в bool

Указатели, целые числа и числа с плавающей точкой могут быть неявно преобразованы в объекты `bool`. Если значение отлично от нуля, результатом неявного преобразования является `true`. В противном случае результат будет равен `false`. Например, значение `int{1}` преобразуется в `true`, а значение `int{}` преобразуется в `false`.

*Указатель на void**

Указатели всегда могут быть неявно преобразованы в `void*`, как показано в листинге 7.10.

Листинг 7.10. Неявное преобразование указателя в void*. Вывод с машины на Windows 10 x64.

```
#include <cstdio>

void print_addr(void* x) {
  printf("0x%p\n", x);
}

int main() {
  int x{};
  print_addr(&x); ❶
  print_addr(nullptr); ❷
}
--------------------------------------------------------------------------------
0x000000F79DCFFB74 ❶
0x0000000000000000 ❷
```

Листинг 7.10 компилируется благодаря неявному преобразованию указателей в `void*`. Функция `print_addr` выводит адрес `x` ❶ и значение `nullptr`, `0` ❷.

Явное преобразование типов

Явные преобразования типов также называются *приведениями*. Первая точка вызова для проведения явного преобразования типов — это инициализация со скобками `{}`. Этот подход имеет главное преимущество в том, что он полностью безопасен и не сужается. Использование фигурной инициализации гарантирует во время компиляции, что разрешены только безопасные, хорошо ведущие себя, не сужающиеся преобразования. Листинг 7.11 показывает пример.

Листинг 7.11. Явное преобразование типов для 4- и 8-байтовых целых чисел

```
#include <cstdio>
#include <cstdint>

int main() {
  int32_t a = 100;
  int64_t b{ a }; ❶
  if (a == b) printf("Non-narrowing conversion!\n"); ❷
  //int32_t c{ b }; // Бах! ❸
}
--------------------------------------------------------------------------------
Non-narrowing conversion! ❷
```

Этот простой пример использует фигурную инициализацию ❶ для построения `int64_t` из `int32_t`. Это правильное преобразование, потому что гарантированно не будет потеряна какая-либо информация. Вы всегда можете хранить 32 бита внутри 64 бит. После корректного преобразования базового типа оригинал всегда будет равен результату (согласно `operator==`).

В примере показывается плохо себя ведущее (сужающееся) преобразование ❸. Компилятор выдаст ошибку. Если не использовать инициализацию с фигурными скобками `{}`, компилятор не жаловался бы, как показано в листинге 7.12.

Листинг 7.12. Рефакторинг из листинга 7.11 без инициализатора со скобками

```
#include <limits>
#include <cstdio>
#include <cstdint>

int main() {
  int64_t b = std::numeric_limits<int64_t>::max();
  int32_t c(b); ❶ // The compiler abides.
  if (c != b) printf("Narrowing conversion!\n"); ❷
}
--------------------------------------------------------------------------------
Narrowing conversion! ❷
```

Здесь осуществляется сужающее преобразование из 64-разрядного целого числа в 32-разрядное целое число ❶. Поскольку значение сужается, выражение `c != b` вы-

числяется как `true` ❷. Такое поведение очень опасно, поэтому в главе 2 рекомендуется использовать как можно больше инициализаций с фигурными скобками.

Приведения в стиле C

Вспомните из главы 6, что именованные функции преобразования позволяют выполнять опасные приведения, которые не разрешены в рамках инициализации. Также можно выполнять приведение в стиле C, но это делается главным образом для поддержания некоторой совместимости между языками. Его использование заключается в следующем:

```
(желаемый-тип)объект-для-приведения
```

Для каждого приведения в стиле C существует некоторое заклинание: `static_casts`, `const_casts` и `reinterpret_casts`, которое обеспечит желаемое преобразование типов. Приведения в стиле C гораздо более опасны, чем именованные (и это говорит о многом).

Синтаксис явных приведений C++ намеренно уродлив и многословен. Это привлекает внимание к точке в коде, где жесткие правила системы типов обходятся или нарушаются. Приведение в стиле C этого не делает. Кроме того, из приведения неясно, какое преобразование имеет в виду программист. При использовании более точных инструментов, таких как именованные приведения, компилятор может, по крайней мере, применять некоторые ограничения. Например, слишком легко забыть правильность `const` при использовании приведения в стиле C, когда предполагалось использование только `reinterpret_cast`.

Предположим, нужно обработать массив `const char*` как беззнаковый в теле функции. Было бы слишком легко написать код, подобный тому, что показан в листинге 7.13.

Листинг 7.13. Крушение поезда в стиле C, которое случайно избавляется от квалификатора const в read_only. (Эта программа имеет неопределенное поведение; вывод осуществляется с машины Windows 10 x64.)

```
#include <cstdio>

void trainwreck(const char* read_only) {
  auto as_unsigned = (unsigned char*)read_only;
  *as_unsigned = 'b'; ❶ // Не сработает в Windows 10 x64
}

int main() {
  auto ezra = "Ezra";
  printf("Before trainwreck: %s\n", ezra);
  trainwreck(ezra);
  printf("After trainwreck: %s\n", ezra);
}
--------------------------------------------------------------------------------
Before trainwreck: Ezra
```

Современные операционные системы применяют шаблоны доступа к памяти. В листинге 7.13 предпринимается попытка записи в память, хранящую строковый литерал `Ezra` ❶. В Windows 10 x64 это приводит к сбою программы с нарушением доступа к памяти (это только постоянная память).

Если попробовать сделать то же самое с `reinterpret_cast`, компилятор выдаст ошибку, как показано в листинге 7.14.

Листинг 7.14. Рефакторинг листинга 7.13 с использованием reinterpret_cast. (Этот код не компилируется.)

```
#include <cstdio>

void trainwreck(const char* read_only) {
  auto as_unsigned = reinterpret_cast<unsigned char*>(read_only); ❶
  *as_unsigned = 'b'; // Crashes on Windows 10 x64
}

int main() {
  auto ezra = "Ezra";
  printf("Before trainwreck: %s\n", ezra);
  trainwreck(ezra);
  printf("After trainwreck: %s\n", ezra);
}
```

Если вы действительно намеревались отбросить правильность `const`, нужно добавить здесь `const_cast` ❶. Код сам задокументировал бы эти намерения и позволил легко найти такие преднамеренные нарушения правил.

Пользовательские приведения типов

В пользовательских типах можно предоставлять пользовательские функции преобразования. Эти функции сообщают компилятору, как ведут себя пользовательские типы при неявном и явном преобразовании типов. Можно объявить эти функции преобразования, используя следующий шаблон использования:

```
struct MyType {
  operator тип-назначения() const {
    // отсюда возвращается тип-назначения.
  --пропуск--
  }
}
```

Например, структура в листинге 7.15 может использоваться как `int` только для чтения.

Листинг 7.15. Класс ReadOnlyInt, содержащий преобразование пользовательского типа в int

```
struct ReadOnlyInt {
  ReadOnlyInt(int val) : val{ val } { }
```

```
  operator int() const { ❶
    return val;
  }
private:
  const int val;
};
```

Метод оператора `int` в ❶ определяет преобразование пользовательского типа *из* `ReadOnlyInt` *в* `int`. Теперь можно использовать типы `ReadOnlyInt`, как и обычные типы `int`, благодаря неявному преобразованию:

```
struct ReadOnlyInt {
  --пропуск--
};
int main() {
  ReadOnlyInt the_answer{ 42 };
  auto ten_answers = the_answer * 10; // int with value 420
}
```

Иногда неявные преобразования могут вызывать удивительное поведение. Всегда стоит попытаться использовать явные преобразования, особенно в случае с пользовательскими типами. Можно добиться явных преобразований с явным ключевым словом `explicit`. Явные конструкторы инструктируют компилятор не рассматривать конструктор как средство неявного преобразования. Те же рекомендации можно предоставить для пользовательских функций преобразования:

```
struct ReadOnlyInt {
  ReadOnlyInt(int val) : val{ val } { }
  explicit operator int() const {
    return val;
  }
private:
  const int val;
};
```

Нужно явно привести `ReadOnlyInt` к `int` с использованием `static_cast`:

```
struct ReadOnlyInt {
  --пропуск--
};
int main() {
  ReadOnlyInt the_answer{ 42 };
  auto ten_answers = static_cast<int>(the_answer) * 10;
}
```

Как правило, этот подход способствует менее неоднозначному коду.

Постоянные выражения

Постоянные выражения — это выражения, которые можно вычислять во время компиляции. По соображениям производительности и безопасности всякий раз,

когда вычисление может быть выполнено во время компиляции, а не во время выполнения, следует это делать. Простые математические операции с использованием литералов являются очевидным примером выражений, которые можно вычислить во время компиляции.

Можно расширить область действия компилятора, используя выражение `constexpr`. Всякий раз, когда вся информация, необходимая для вычисления выражения, присутствует во время компиляции, компилятор *вынужден это делать*, если это выражение помечено как `constexpr`. Это простое обязательство может оказать удивительно большое влияние на читаемость кода и производительность во время выполнения.

И `const`, и `constexpr` тесно связаны между собой. Принимая во внимание, что `constexpr` обеспечивает, что выражение вычисляется во время компиляции, `const` обеспечивает, что переменная не может изменяться в некоторой области (во время выполнения). Все выражения `constexpr` являются `const`, потому что они всегда имеют фиксированное значение во время выполнения.

Все выражения `constexpr` начинаются с одного или нескольких основных типов (`int`, `float`, `wchar_t` и т. д.). Можно построить выражение на основе этих типов, используя операторы и функции `constexpr`. Постоянные выражения используются главным образом для замены вычисленных вручную значений в коде. Обычно это создает код, который является более надежным и простым для понимания, поскольку можно исключить так называемые *магические значения* — когда вычисленные вручную константы копируются и вставляются непосредственно в исходный код.

Красочный пример

Рассмотрим следующий пример, где некоторые библиотеки используют объекты `Color`, закодированные с использованием представления цвета-насыщенности-значения (hue-saturation-value, HSV):

```
struct Color {
  float H, S, V;
};
```

Грубо говоря, цвет соответствует семейству цветов, таких как красный, зеленый или оранжевый. Насыщенность соответствует красочности или интенсивности. Значение соответствует яркости цвета.

Предположим, нужно создать экземпляр объекта `Color` с помощью красно-зелено-синего (red-green-blue, RGB) представления. Можно использовать конвертер для расчета RGB из HSV вручную, но это яркий пример, где можно использовать `constexpr` для устранения магических значений. Прежде чем вы сможете написать функцию преобразования, нужно определить несколько вспомогательных функций, а именно `min`, `max` и `modulo`. Листинг 7.16 реализует эти функции.

Листинг 7.16. Несколько функций constexpr для управления объектами uint8_t

```
#include <cstdint>
constexpr uint8_t max(uint8_t a, uint8_t b) { ❶
  return a > b ? a : b;
}
constexpr uint8_t max(uint8_t a, uint8_t b, uint8_t c) {❷
  return max(max(a, b), max(a, c));
}
constexpr uint8_t min(uint8_t a, uint8_t b) { ❸
  return a < b ? a : b;
}
constexpr uint8_t min(uint8_t a, uint8_t b, uint8_t c) { ❹
  return min(min(a, b), min(a, c));
}
constexpr float modulo(float dividend, float divisor) { ❺
  const auto quotient = dividend / divisor; ❻
  return divisor * (quotient - static_cast<uint8_t>(quotient));
}
```

Каждая функция помечена как `constexpr`, что говорит компилятору, что функция должна быть вычисляемой во время компиляции. Функция `max` ❶ использует тернарный оператор, чтобы вернуть наибольшее значение аргумента. Версия `max` с тремя аргументами ❷ использует переходное свойство сравнения; оценив `max` с двумя аргументами для пар `a, b` и `a, c`, можно найти максимум этого промежуточного результата, чтобы определить общий максимум. Поскольку версия `max` с двумя аргументами является `constexpr`, это абсолютно законно.

ПРИМЕЧАНИЕ

Нельзя использовать `fmax` из заголовка `<math.h>` по той же причине: это не `constexpr`.

Версии `min` ❸❹ делают то же самое, но с очевидной модификацией — сравнение перевернуто. Функция `modulo` ❺ является быстрой и грязной `constexpr`-версией функции `fmod` в C, которая вычисляет остаток с плавающей точкой от деления первого аргумента (`divisor`) на второй аргумент (`dividend`). Поскольку `fmod` *не* является `constexpr`, вы сделали свой собственный выбор. Сначала вычисляется частное ❻. Затем вычитается неотъемлемая часть частного, используя `static_cast` и вычитание. Умножение десятичной части частного на `divisor` дает результат.

Имея в своем арсенале набор служебных функций `constexpr`, теперь вы можете реализовать функцию преобразования `rgb_to_hsv`, как показано в листинге 7.17.

Листинг 7.17. Функция преобразования constexpr из RGB в HSV

```
--пропуск--
constexpr Color rgb_to_hsv(uint8_t r, uint8_t g, uint8_t b) {
  Color c{}; ❶
  const auto c_max = max(r, g, b);
  c.V = c_max / 255.0f; ❷

  const auto c_min = min(r, g, b);
```

```
  const auto delta = c.V - c_min / 255.0f;
  c.S = c_max == 0 ? 0 : delta / c.V; ❸

  if (c_max == c_min) { ❹
    c.H = 0;
    return c;
  }
  if (c_max == r) {
    c.H = (g / 255.0f - b / 255.0f) / delta;
  } else if (c_max == g) {
    c.H = (b / 255.0f - r / 255.0f) / delta + 2.0f;
  } else if (c_max == b) {
    c.H = (r / 255.0f - g / 255.0f) / delta + 4.0f;
  }
  c.H *= 60.0f;
  c.H = c.H >= 0.0f ? c.H : c.H + 360.0f;
  c.H = modulo(c.H, 360.0f); ❺
  return c;
}
```

Здесь объявляется и инициализируется `Color c` ❶, который в конечном итоге будет возвращен функцией `rgb_to_hsv`. Значение `Color`, `V` вычисляется в ❷ путем масштабирования максимального значения r, g и b. Затем насыщенность S определяется путем вычисления расстояния между минимальным и максимальным значениями RGB и масштабирования на `V` ❸. Если вы представляете значения HSV как существующие внутри цилиндра, *насыщенность* — это расстояние по горизонтальной оси, а *значение* — это расстояние по вертикальной оси. *Оттенок* — это угол. Для краткости я не буду вдаваться в подробности о том, как этот угол вычисляется, но вычисление выполняется между ❹ и ❺. По сути, это влечет за собой вычисление угла как смещения от угла доминирующего компонента цвета. Все это масштабируется и модулируется, чтобы соответствовать интервалу от 0 до 360 градусов, и сохраняется в `H`. Наконец, возвращается `c`.

ПРИМЕЧАНИЕ

Для объяснения формулы, используемой для преобразования HSV в RGB, см. en.wikipedia.org/wiki/HSL_and_HSV#Color_conversion_formulae.

Здесь происходит довольно много вычислений, но все они производятся во время компиляции. Это означает, что при инициализации цветов компилятор инициализирует `Color` со всеми заполненными плавающими полями HSV:

```
--пропуск--
int main() {
  auto black = rgb_to_hsv(0,     0,   0);
  auto white = rgb_to_hsv(255, 255, 255);
  auto red   = rgb_to_hsv(255,   0,   0);
  auto green = rgb_to_hsv(  0, 255,   0);
  auto blue  = rgb_to_hsv(  0,   0, 255);
  // TODO: Вывести здесь результат.
}
```

Вы указали компилятору, что каждое из этих значений цвета оценивается во время компиляции. В зависимости от того как использовать эти значения в остальной части программы, компилятор может решить, вычислять их во время компиляции или во время выполнения. В результате компилятор обычно может выдавать инструкции с жестко закодированными *магическими числами*, соответствующими значениям HSV для каждого цвета.

Использование constexpr

Существуют некоторые ограничения на то, какие функции могут быть `constexpr`, но эти ограничения ослабляются с каждой новой версией C++.

В определенных контекстах, таких как встроенная разработка, `constexpr` незаменим. Если выражение может быть объявлено `constexpr`, настоятельно рекомендуется сделать это. Использование `constexpr` вместо вычисляемых вручную литералов может сделать код более выразительным. Часто это также может серьезно повысить производительность и безопасность во время выполнения.

Изменчивые выражения

Ключевое слово `volatile` сообщает компилятору, что каждый доступ, полученный с помощью этого выражения, должен рассматриваться как видимый побочный эффект. Это означает, что доступ не может быть оптимизирован или переупорядочен с другим видимым побочным эффектом. Это ключевое слово имеет решающее значение в некоторых разработках, таких как встроенное программирование, где чтение и запись в некоторые специальные части памяти влияют на базовую систему. Ключевое слово `volatile` не позволяет компилятору оптимизировать такой доступ. Листинг 7.18 показывает, почему может понадобиться ключевое слово `volatile`, содержащее инструкции, которые компилятор обычно оптимизирует.

Листинг 7.18. Функция, содержащая мертвое хранилище и избыточную загрузку

```
int foo(int& x) {
  x = 10; ❶
  x = 20; ❷
  auto y = x; ❸
  y = x; ❹
  return y;
}
```

Поскольку `x` назначен как ❶, но никогда не использовался до переназначения ❷, он называется *мертвым хранилищем* и является прямым кандидатом для оптимизации. Есть похожая история, где `x` используется для установки значения `y` дважды без каких-либо промежуточных инструкций ❸❹. Это называется *избыточной нагрузкой* и также является кандидатом на оптимизацию.

Можно ожидать, что любой приличный компилятор оптимизирует предыдущую функцию во что-то похожее на листинг 7.19.

Листинг 7.19. Правдоподобная оптимизация листинга 7.18

```
int foo(int& x) {
  x = 20;
  return x;
}
```

В некоторых разработках избыточное чтение и мертвые хранилища могут иметь видимые побочные эффекты в системе. Добавляя ключевое слово `volatile` в аргумент `foo`, можно избежать того, чтобы оптимизатор избавился от этих важных обращений, как показано в листинге 7.20.

Листинг 7.20. Модификация volatileTODO листинга 7.18

```
int foo(volatile int& x) {
  x = 10;
  x = 20;
  auto y = x;
  y = x;
  return y;
}
```

Теперь компилятор будет выдавать инструкции для выполнения каждой запрограммированной операции чтения и записи.

Распространенным заблуждением является то, что `volatile` имеет отношение к параллельному программированию. Это не так. Переменные, помеченные как `volatile`, обычно не являются поточно-ориентированными. Во второй части книги обсуждается `std::atomic`, который гарантирует определенные поточно-ориентированные примитивы для типов. Слишком часто `volatile` путается с `atomic`!

Итоги

В этой главе рассматриваются основные функции операторов, которые являются основными единицами работы в программе. Вы изучили несколько аспектов преобразования типов и взяли под контроль динамическое управление памятью из среды. Вы также познакомились с выражениями `constexpr/volatile`. С этими инструментами в руках можно выполнять практически любые задачи системного программирования.

Упражнения

7.1. Создайте класс `UnsignedBigInteger`, который может обрабатывать числа больше, чем `long`. Можно использовать байтовый массив в качестве внутреннего представления (например, `uint8_t[]` или `char[]`). Реализуйте перегрузки операторов для оператора + и оператора -. Выполните проверки на переполнения во время выполнения. Для `intrepid` также исполь-

зуйте `operator *`, `operator /` и `operator %`. Убедитесь, что перегрузки оператора работают как для типов `int`, так и для типов `UnsignedBigInteger`. Реализуйте преобразование типа `operator int`. Выполните проверку на сужение во время выполнения.

7.2. Создайте класс `LargeBucket`, который может хранить до 1 МБ данных. Расширьте класс `Heap` так, чтобы он выдавал `LargeBucket` для выделений, превышающих 4096 байт. Удостоверьтесь, что `std::bad_alloc` по-прежнему генерируется всякий раз, когда куча не может выделить контейнер соответствующего размера.

Что еще почитать?

- Международный стандарт ИСО/МЭК (2017) — Язык программирования C++ (Международная организация по стандартизации; Женева, Швейцария; isocpp.org/std/the-standard/)

8 Инструкции

Прогресс двигают не те, кто рано встает, его стимулируют лентяи, старающиеся облегчить себе жизнь.

Роберт А. Хайнлайн, «Достаточно времени для любви»

Каждая функция C++ содержит последовательность *операторов*, которые являются программными конструкциями, определяющими порядок выполнения. В этой главе используется понимание жизненного цикла объекта, шаблонов и выражений для изучения нюансов операторов.

Инструкции-выражения

Инструкция-выражение — это выражение, за которым следует точка с запятой (;). Инструкции-выражения составляют большинство операторов в программе. Можно превратить любое выражение в оператор, что стоит делать всякий раз, когда нужно вычислить выражение, но отбросить результат. Конечно, это полезно только в том случае, если при вычислении этого выражения возникает побочный эффект, такой как вывод в консоль или изменение состояния программы.

Листинг 8.1 содержит несколько операторов-выражений.

Листинг 8.1. Простая программа, содержащая несколько операторов-выражений

```
#include <cstdio>

int main() {
  int x{};
  ++x; ❶
  42; ❷
  printf("The %d True Morty\n", x); ❸
}
-------------------------------------------------------------------------------
The 1 True Morty ❸
```

Оператор-выражение в ❶ имеет побочный эффект (увеличение `x`), а в ❷ — нет. Оба действительны (хотя тот, что в ❷, бесполезен). Вызов функции `printf` ❸ также является оператором-выражением.

Составные операторы

Составные операторы, также называемые *блоками*, представляют собой последовательность операторов, заключенную в фигурные скобки `{}`. Блоки полезны в управляющих структурах, таких как операторы `if`, потому что может потребоваться выполнить несколько операторов вместо одного.

Каждый блок объявляет новую область видимости, которая называется *областью видимости блока*. Из главы 4 вы знаете, что объекты с автоматической длительностью хранения, объявленной в области видимости блока, имеют время жизни, связанное с блоком. Переменные, объявленные в блоке, уничтожаются в четко определенном порядке: обратном тому, в котором были объявлены.

В листинге 8.2 используется надежный класс `Tracer` из листинга 4.5 для изучения области видимости блока.

Листинг 8.2. Программа, исследующая составные операторы с классом Tracer

```
#include <cstdio>

struct Tracer {
  Tracer(const char* name) : name{ name } {
    printf("%s constructed.\n", name);
  }
  ~Tracer() {
    printf("%s destructed.\n", name);
  }
private:
  const char* const name;
};

int main() {
  Tracer main{ "main" }; ❶
  {
    printf("Block a\n"); ❷
    Tracer a1{ "a1" }; ❸
    Tracer a2{ "a2" }; ❹
  }
  {
    printf("Block b\n"); ❺
    Tracer b1{ "b1" }; ❻
    Tracer b2{ "b2" }; ❼
  }
}
--------------------------------------------------------------------------------
main constructed. ❶
Block a ❷
```

```
a1 constructed. ❸
a2 constructed. ❹
a2 destructed.
a1 destructed.
Block b ❺
b1 constructed. ❻
b2 constructed. ❼
b2 destructed.
b1 destructed.
main destructed.
```

Листинг 8.2 начинается с инициализации `Tracer` под названием `main` ❶. Далее генерируются два составных оператора. Первый составной оператор начинается с левой фигурной скобки {, за которой следует первый оператор блока, который выводит `Block a` ❷. Создаются два `Tracer`, `a1` ❸ и `a2` ❹, а затем блок закрывается правой скобкой }. Эти два экземпляра уничтожаются, когда выполнение проходит через `Block a`. Обратите внимание, что они уничтожаются в порядке, обратном их инициализации: `a2`, затем `a1`.

Также обратите внимание на другой составной оператор после блока `a`, где выводится `Block b` ❺, а затем создаются два экземпляра `Tracer`, `b1` ❻ и `b2` ❼. Их поведение идентично: сначала уничтожается `b2`, а затем `b1`. Как только выполнение проходит через `Block b`, область видимости `main` завершается и `Tracer main` окончательно уничтожается.

Операторы объявлений

Операторы объявлений (или просто *объявления*) вводят идентификаторы, такие как функции, шаблоны и пространства имен, в программы. В этом разделе рассматриваются некоторые новые функции этих знакомых объявлений, а также псевдонимы типов, атрибуты и структурированные привязки.

ПРИМЕЧАНИЕ

Выражение `static_assert`, о котором вы узнали в главе 6, также является оператором объявления.

Функции

Объявление функции, также называемое *сигнатурой*, или *прототипом* функции, определяет входные и выходные значения функции. Декларация не должна включать имена параметров, только их типы. Например, следующая строка объявляет функцию `randomize`, которая принимает ссылку на `uint32_t` и возвращает `void`:

```
void randomize(uint32_t&);
```

Функции, которые *не являются функциями-членами*, иногда называются *свободными функциями*, и они всегда объявляются за пределами `main()` в области пространства имен. *Определение функции* включает в себя объявление функции, а также ее тело. Объявление функции определяет интерфейс функции, тогда как определение функции определяет ее реализацию. Например, следующее определение является одной из возможных реализаций функции `randomize`:

```
void randomize(uint32_t& x) {
  x = 0x3FFFFFFF & (0x41C64E6D * x + 12345) % 0x80000000;
}
```

ПРИМЕЧАНИЕ

Такая реализация `randomize` является линейным конгруэнтным генератором — генератором случайных чисел примитивного типа. Дополнительные источники о генерации случайных чисел см. в разделе «Что еще почитать?» на с. 310.

Как вы, наверное, заметили, объявления функций являются необязательными. Так почему же они существуют?

Ответ заключается в том, что можно использовать объявленные функции во всем коде, если они где-то определены. Цепочка инструментов вашего компилятора может понять это. (Как это работает, вы узнаете в главе 21.)

Программа в листинге 8.3 определяет, сколько итераций требуется генератору случайных чисел, чтобы получить числа 0x4c4347 и 0x474343.

Листинг 8.3. Программа, использующая функцию main, которая не будет определена позже

```
#include <cstdio>
#include <cstdint>

void randomize(uint32_t&); ❶

int main() {
  size_t iterations{}; ❷
  uint32_t number{ 0x4c4347 }; ❸
  while (number != 0x474343) { ❹
    randomize(number); ❺
    ++iterations; ❻
  }
  printf("%zu", iterations); ❼
}

void randomize(uint32_t& x) {
  x = 0x3FFFFFFF & (0x41C64E6D * x + 12345) % 0x80000000; ❽
}
--------------------------------------------------------------------------------
927393188 ❼
```

Во-первых, объявляется `randomize` ❶. Внутри `main` инициализируется переменная счетчика итераций, равная нулю ❷, и переменная `number` — 0x4c4347 ❸. Цикл `while` проверяет, равно ли `number` целевому 0x4c4347 ❹. Если это не так, вызывается `randomize` ❺ и увеличивается `iterations` ❻. Обратите внимание, что `randomize` еще не определена. Как только `number` будет равна целевому числу, выведется количество `iterations` ❼, прежде чем завершить работу `main`. Наконец, определяется `randomize` ❽. Вывод программы показывает, что для случайного получения целевого значения требуется почти миллиард итераций.

Попробуйте удалить определение `randomize` и перекомпилировать фрагмент. Вы должны получить сообщение о том, что определение `randomize` не может быть найдено.

Также можно отделить объявления методов от их определений. Как и в случае с функциями, не являющимися членами, можно объявить метод, опуская его тело. Например, класс `RandomNumberGenerator` ниже заменяет функцию `randomize` следующим:

```
struct RandomNumberGenerator {
  explicit RandomNumberGenerator(uint32_t seed) ❶
    : number{ seed } {} ❷
  uint32_t next(); ❸
private:
  uint32_t number;
};
```

Можно создать `RandomNumberGenerator` с значением `seed` ❶, которое он использует для инициализации переменной-члена `number` ❷. Следующая функция объявлена с использованием тех же правил, что и для функций, не являющихся членами ❸. Чтобы обеспечить определение следующего кода, нужно использовать оператор разрешения контекста и имя класса, чтобы понять, какой метод необходимо определить. В противном случае определение метода аналогично определению функции, не являющейся членом:

```
uint32_t❶ RandomNumberGenerator::❷next() {
  number = 0x3FFFFFFF & (0x41C64E6D * number + 12345) % 0x80000000; ❸
  return number; ❹
}
```

Это определение имеет тот же тип возврата, что и объявление ❶. `RandomNumberGenerator::construct` указывает, что определяется метод ❷. Детали функции в основном те же самые ❸, за исключением того что возвращается копия состояния генератора случайных чисел, а не ссылка на параметр ❹.

В листинге 8.4 показано, как можно переписать листинг 8.3 для добавления `RandomNumberGenerator`.

Как и в листинге 8.3, объявление было отделено от определения. После объявления конструктора, который инициализирует элемент `iterations` нулем ❶ и задает его член `number` как `seed` ❷, объявления методов `next` ❸ и `get_iterations` ❹ содержат реализации. В пределах `main` инициализируется класс `RandomNumberGenerator` с за-

данным начальным значением 0x4c4347 ❺ и вызывается метод `next` для извлечения новых случайных чисел ❻. Результаты одинаковы ❼. Как и раньше, определения `next` и `get_iterations` следуют за их использованием в `main` ❽❾.

Листинг 8.4. Рефакторинг листинга 8.3 с использованием класса RandomNumberGenerator

```
#include <cstdio>
#include <cstdint>
struct RandomNumberGenerator {
  explicit RandomNumberGenerator(uint32_t seed)
    : iterations{}❶, number { seed }❷ {}
  uint32_t next(); ❸
  size_t get_iterations() const; ❹
private:
  size_t iterations;
  uint32_t number;
};

int main() {
  RandomNumberGenerator rng{ 0x4c4347 }; ❺
  while (rng.next() != 0x474343) { ❻
    // Не делать ничего...
  }
  printf("%zu", rng.get_iterations()); ❼
}

uint32_t RandomNumberGenerator::next() { ❽
  ++iterations;
  number = 0x3FFFFFFF & (0x41C64E6D * number + 12345) % 0x80000000;
  return number;
}

size_t RandomNumberGenerator::get_iterations() const { ❾
  return iterations;
}
--------------------------------------------------------------------------------
927393188 ❼
```

ПРИМЕЧАНИЕ

Полезность разделения определения и объявления может быть неочевидной, поскольку мы до сих пор работали с программами из одного исходного файла. В главе 21 рассматриваются программы с несколькими исходными файлами, в которых разделение объявлений и определений дает основные преимущества.

Пространства имен

Пространства имен предотвращают конфликты имен. В больших проектах или при импорте библиотек пространства имен необходимы для устранения неоднозначности нужных символов.

Размещение символов в пространствах имен

По умолчанию все объявленные символы попадают в *глобальное пространство имен*. Глобальное пространство имен содержит все символы, к которым можно обращаться, не добавляя квалификаторы пространства имен. Помимо нескольких классов в пространстве имен `std` мы использовали объекты, живущие исключительно в глобальном пространстве имен.

Чтобы поместить символ в пространство имен, отличное от глобального пространства имен, объявите символ в *блоке пространства имен*. Блок пространства имен имеет следующую форму:

```
namespace BroopKidron13 {
  // Все символы, объявленные в этом блоке,
  // принадлежат пространству имен BroopKidron13
}
```

Пространства имен могут быть вложены друг в друга одним из двух способов. Во-первых, можно просто вложить блоки пространства имен:

```
namespace BroopKidron13 {
  namespace Shaltanac {
    // Все символы, объявленные в этом блоке,
    // принадлежат пространству имен BroopKidron13::Shaltanac
  }
}
```

Во-вторых, можно использовать оператор разрешения контекста:

```
namespace BroopKidron13::Shaltanac {
  // Все символы, объявленные в этом блоке,
  // принадлежат пространству имен BroopKidron13::Shaltanac
}
```

Последний подход более лаконичен.

Использование символов в пространствах имен

Чтобы использовать символ в пространстве имен, всегда можно применить оператор разрешения контекста для указания полного имени символа. Это позволяет предотвратить конфликты имен в больших проектах или при использовании сторонней библиотеки. Если вы и другой программист используете один и тот же символ, можно избежать двусмысленности, поместив символ в пространство имен.

Листинг 8.5 показывает, как можно использовать полные имена символов для доступа к символу в пространстве имен.

Листинг 8.5. Вложенные блоки пространства имен с использованием оператора разрешения контекста

```
#include <cstdio>

namespace BroopKidron13::Shaltanac { ❶
  enum class Color { ❷
```

```
    Mauve,
    Pink,
    Russet
  };
}

int main() {
  const auto shaltanac_grass{ BroopKidron13::Shaltanac::Color::Russet❸ };
  if(shaltanac_grass == BroopKidron13::Shaltanac::Color::Russet) {
    printf("The other Shaltanac's joopleberry shrub is always "
           "a more mauvey shade of pinky russet.");
  }
}
--------------------------------------------------------------------------------
The other Shaltanac's joopleberry shrub is always a more mauvey shade of pinky russet.
```

В листинге 8.5 используются вложенные пространства имен ❶ и объявляется тип `Color` ❷. Чтобы использовать `Color`, применяется оператор разрешения контекста для указания полного имени символа, `BroopKidron13::Shaltanac::Color`. Поскольку `Color` — это `enumclass`, оператор разрешения контекста используется для доступа к его значениям, как, например, при присваивании `Russet` значения `shaltanac_grass` ❸.

Использование директив

Вы можете использовать *директиву* `using`, чтобы не печатать много кода. Директива `using` импортирует символ в блок или, если она была объявлена в области пространства имен, в текущее пространство имен. В любом случае, необходимо ввести полный путь к пространству имен только один раз. Использование имеет следующий шаблон:

```
using мой-тип;
```

Соответствующий *мой-тип* импортируется в текущее пространство имен или блок, что означает, что больше не нужно использовать его полное имя. В листинге 8.6 рефакторинг листинга 8.5 с использованием директивы `using`.

Листинг 8.6. Рефакторинг листинга 8.5, использующий директиву using

```
#include <cstdio>

namespace BroopKidron13::Shaltanac {
  enum class Color {
    Mauve,
    Pink,
    Russet
  };
}

int main() {
  using BroopKidron13::Shaltanac::Color; ❶
  const auto shaltanac_grass = Color::Russet❷;
  if(shaltanac_grass == Color::Russet❸) {
    printf("The other Shaltanac's joopleberry shrub is always "
```

```
            "a more mauvey shade of pinky russet.");
  }
}
--------------------------------------------------------------------------------
The other Shaltanac's joopleberry shrub is always a more mauvey shade of pinky russet.
```

С помощью директивы `using` ❶ внутри `main` больше не нужно вводить пространство имен `BroopKidron13::Shaltanac`, чтобы использовать `Color` ❷❸.

При достаточной осторожности можно ввести все символы из данного пространства имен в глобальное пространство имен с помощью директивы `using namespace`.

В листинге 8.7 подробно описан листинг 8.6: пространство имен `BroopKidron13::Shaltanac` содержит несколько символов, которые нужно импортировать в глобальное пространство имен, чтобы избежать излишней типизации.

Листинг 8.7. Рефакторинг листинга 8.6 с несколькими символами, импортированными в глобальное пространство имен

```
#include <cstdio>

namespace BroopKidron13::Shaltanac {
  enum class Color {
    Mauve,
    Pink,
    Russet
  };

  struct JoopleberryShrub {
    const char* name;
    Color shade;
  };

  bool is_more_mauvey(const JoopleberryShrub& shrub) {
    return shrub.shade == Color::Mauve;
  }
}

using namespace BroopKidron13::Shaltanac; ❶

int main() {
  const JoopleberryShrub❷ yours{
    "The other Shaltanac",
    Color::Mauve❸
  };

  if (is_more_mauvey(yours)❹) {
    printf("%s's joopleberry shrub is always a more mauvey shade of pinky"
           "russet.", yours.name);
  }
}
--------------------------------------------------------------------------------
The other Shaltanac's joopleberry shrub is always a more mauvey shade of pinky russet.
```

С помощью директивы `using namespace` ❶ можно использовать классы ❷, классы перечислений ❸, функции ❹ и т. д. без необходимости вводить полностью определенные имена. Конечно, нужно быть очень осторожными с добавлением существующих типов в глобальное пространство имен. Как правило, это плохо, если в одной единице трансляции появилось слишком много директив пространства имен.

ПРИМЕЧАНИЕ

Никогда не следует помещать директиву `using namespace` в заголовочный файл. Каждый исходный файл, содержащий ваш заголовок, будет сбрасывать все символы из директивы `using` в глобальное пространство имен. Это может вызвать проблемы, которые очень трудно отладить.

Совмещение имен типов

Псевдоним типа определяет имя, которое ссылается на ранее определенное имя. Можно использовать псевдоним типа как синоним существующего имени типа.

Нет никакой разницы между типом и любыми псевдонимами типа, ссылающимися на него. Кроме того, псевдонимы типов не могут изменить значение существующего имени типа.

Чтобы объявить псевдоним типа, используйте следующий формат, где используется имя *псевдонима-типа* и *целевой-тип*:

```
using псевдоним-типа = целевой-тип;
```

В листинге 8.8 используются псевдонимы двух типов: `String` и `ShaltanacColor`.

Листинг 8.8. Рефакторинг листинга 8.7 с псевдонимом типа

```
#include <cstdio>

namespace BroopKidron13::Shaltanac {
  enum class Color {
    Mauve,
    Pink,
    Russet
  };
}

using String = const char[260]; ❶
using ShaltanacColor = BroopKidron13::Shaltanac::Color; ❷

int main() {
  const auto my_color{ ShaltanacColor::Russet }; ❸
  String saying { ❹
    "The other Shaltanac's joopleberry shrub is "
    "always a more mauvey shade of pinky russet."
  };
  if (my_color == ShaltanacColor::Russet) {
    printf("%s", saying);
  }
}
```

Листинг 8.8 объявляет псевдоним типа `String`, который ссылается на `const char [260]` ❶. В этом листинге также объявляется псевдоним типа `ShaltanacColor`, который ссылается на `BroopKidron13::Shaltanac::Color` ❷. Можно использовать эти псевдонимы типа в качестве замены для очистки кода. В `main ShaltanacColor` используется, чтобы удалить все вложенные пространства имен ❸ и `String`, чтобы сделать объявление `saying` «чище» ❹.

ПРИМЕЧАНИЕ

Псевдонимы типов могут появляться в любой области: блоке, классе или пространстве имен.

Можно ввести параметры шаблона в псевдонимы типов. Это позволяет использовать их в двух важных случаях:

- можно выполнить частичное применение параметров шаблона. *Частичное применение* — это процесс фиксации некоторого количества аргументов в шаблоне при создании другого шаблона с меньшим количеством параметров;
- можно определить псевдоним типа для шаблона с полностью заданным набором параметров.

Написание экземпляров шаблонов может быть долгим делом, и избежать туннельного синдрома помогают псевдонимы типов.

В листинге 8.9 объявляется класс `NarrowCaster` с двумя параметрами шаблона. Затем используется псевдоним типа, чтобы частично применить один из его параметров и создать новый тип.

Листинг 8.9. Частичное применение класса NarrowCaster с использованием псевдонима типа

```
#include <cstdio>
#include <stdexcept>

template <typename To, typename From>
struct NarrowCaster const { ❶
  To cast(From value) {
    const auto converted = static_cast<To>(value);
    const auto backwards = static_cast<From>(converted);
    if (value != backwards) throw std::runtime_error{ "Narrowed!" };
    return converted;
  }
};

template <typename From>
using short_caster = NarrowCaster<short, From>; ❷

int main() {
  try {
    const short_caster<int> caster; ❸
    const auto cyclic_short = caster.cast(142857);
    printf("cyclic_short: %d\n", cyclic_short);
```

```
  } catch (const std::runtime_error& e) {
    printf("Exception: %s\n", e.what()); ❹
  }
}
-------------------------------------------------
Exception: Narrowed! ❹
```

Во-первых, реализуется шаблонный класс `NarrowCaster`, который имеет те же функциональные возможности, что и шаблон функции `small_cast` в листинге 6.6 (на с. 216): он будет выполнять `static_cast`, а затем проверять сужение ❶. Затем объявляется псевдоним типа `short_caster`, который частично применяет `short` как тип `To` в `NarrowCast`. Внутри `main` объявляется объект `caster` типа `short_caster<int>` ❸. Один параметр шаблона в псевдониме типа `short_caster` применяется к оставшемуся параметру типа из псевдонима типа — `From` ❷. Другими словами, тип `short_cast<int>` является синонимом `NarrowCaster<short, int>`. В конце концов, результат тот же: с 2-байтовым коротким получается сужающее исключение при попытке привести `int` со значением 142857 к `short` ❹.

Структурированные привязки

Структурированные привязки позволяют распаковывать объекты на их составные элементы. Любой тип, чьи нестатические члены-данные являются публичными, может быть распакован таким образом: например, типы POD (классы простых данных), представленные в главе 2. *Синтаксис структурированной привязки* выглядит следующим образом:

```
auto [объект-1, объект-2, ...] = простые-данные;
```

Эта строка инициализирует произвольное количество объектов (*объект-1, объект-2* и т. д.), снимая их один за другим с объекта POD. Объекты отслаиваются от POD сверху вниз и заполняют структурированную привязку слева направо. Рассмотрим функцию `read_text_file`, которая принимает строковый аргумент, соответствующий пути к файлу. Такая функция может не работать, например, если файл заблокирован или не существует. Есть два варианта обработки ошибок:

- можно вызвать исключение в файле `read_text_file`;
- можно вернуть код статуса успеха из функции.

Давайте рассмотрим второй вариант.

Тип POD в листинге 8.10 будет служить точным типом возврата из функции `read_text_file`.

Листинг 8.10. Тип TextFile, который будет возвращен функцией read_text_file

```
struct TextFile {
  bool success; ❶
  const char* contents; ❷
  size_t n_bytes; ❸
};
```

Во-первых, флаг сообщает вызывающей стороне, был ли вызов функции успешным ❶. Далее следуют содержимое `file` ❷ и его размер `n_bytes` ❸.

Прототип `read_text_file` выглядит так:

```
TextFile read_text_file(const char* path);
```

Можно использовать объявление структурированной привязки для разбиения `TextFile` на части в программе, как показано в листинге 8.11.

Листинг 8.11. Программа, имитирующая чтение текстового файла, который возвращает POD, использующийся в структурированной привязке

```
#include <cstdio>

struct TextFile { ❶
  bool success;
  const char* data;
  size_t n_bytes;
};

TextFile read_text_file(const char* path) { ❷
  const static char contents[]{ "Sometimes the goat is you." };
  return TextFile{
    true,
    contents,
    sizeof(contents)
  };
}

int main() {
  const auto [success, contents, length]❸ = read_text_file("REAMDE.txt"); ❹
  if (success❺) {
    printf("Read %zu bytes: %s\n", length❻, contents❼);
  } else {
    printf("Failed to open REAMDE.txt.");
  }
}
--------------------------------------------------------------------------------
Read 27 bytes: Sometimes the goat is you.
```

`TextFile` был объявлен ❶, а затем было предоставлено фиктивное определение для `read_text_file` ❷. (На самом деле она не читает файл; подробнее об этом в части 2.)

В `main` вызывается `read_text_file` ❹ и используется объявление структурированной привязки, чтобы распаковать результаты в три различные переменные: `success`, `content` и `length` ❸. После структурированной привязки можно использовать все эти переменные, как если бы они были объявлены по отдельности ❺❻❼.

ПРИМЕЧАНИЕ

Типы в объявлении структурированной привязки необязательно должны совпадать.

Атрибуты

Атрибуты применяют определяемые реализацией функции к оператору-выражению. Атрибуты вводятся с помощью двойных скобок [[]], содержащих список из одного или нескольких разделенных запятыми элементов атрибута.

В табл. 8.1 перечислены стандартные атрибуты.

Таблица 8.1. Стандартные атрибуты

Атрибут	Значение
`[[noreturn]]`	Указывает, что функция ничего не возвращает
`[[deprecated("причина")]]`	Указывает, что это выражение устарело; то есть его использование не рекомендуется. "Причина" является необязательной и указывает причину устаревания
`[[fallthrough]]`	Указывает, что выполнение случая в операторе `switch` намеревается перейти к следующему случаю. Это позволяет избежать ошибок компиляции, которые будут проверять падение случая в `switch`, потому что это редко происходит
`[[nodiscard]]`	Указывает, что следует использовать следующую функцию или объявление типа. Если код, использующий этот элемент, отбрасывает значение, компилятор должен выдать предупреждение
`[[maybe_unused]]`	Указывает, что следующий элемент может быть не использован и компилятор не должен предупреждать об этом
`[[carries_dependency]]`	Используется в заголовке `<atomic>`, чтобы помочь компилятору оптимизировать определенные операции с памятью. Вы вряд ли столкнетесь с этим напрямую

В листинге 8.12 показано использование атрибута `[[noreturn]]` путем определения функции, которая никогда не возвращает значения.

Листинг 8.12. Программа, где показано использование атрибута [[noreturn]]

```
#include <cstdio>
#include <stdexcept>

[[noreturn]] void pitcher() { ❶
  throw std::runtime_error{ "Knuckleball." }; ❷
}

int main() {
  try {
    pitcher(); ❸
  } catch(const std::exception& e) {
    printf("exception: %s\n", e.what()); ❹
  }
}
-------------------------------------------------------------------------------
Exception: Knuckleball. ❹
```

Сначала объявляется функция `pitcherc` атрибутом `[[noreturn]]` ❶. В этой функции генерируется исключение ❷. Поскольку исключение генерируется всегда, `pitcher` никогда не возвращает значения (следовательно, подойдет атрибут `[[noreturn]]`). В `main` вызывается `pitcher` ❸ и обрабатывается пойманное исключение ❹. Конечно, этот листинг работает и без атрибута `[[noreturn]]`, но предоставление этой информации компилятору дает более полное понимание кода (и потенциально позволяет оптимизировать программу).

Ситуации, в которых нужно использовать атрибут, редки, однако они передают компилятору полезную информацию.

Операторы выбора

Операторы выбора выражают условный поток управления. Существуют два варианта операторов выбора — `if` и `switch`.

Оператор if

Оператор `if` имеет знакомую форму, показанную в листинге 8.13.

Листинг 8.13. Синтаксис оператора if

```
if (условие-1) {
  // Выполняется, только если условие-1 равно true ❶
} else if (условие-2) { // необязательно
  // Выполняется только если условие-2 равно true ❷
}
// ... столько else if, сколько потребуется
--пропуск--
} else { // необязательно
  // Выполняется, только если ни одно из условий не равно true ❸
}
```

При встрече с оператором `if` сначала вычисляется выражение `условия-1`. Если оно равно `true`, блок в ❶ выполняется и оператор `if` перестает выполняться (ни один из остальных операторов `if` или `else` не рассматривается). Если это не так, условия операторов `if`вычисляются по порядку. Это не обязательно, и можно поставить столько условий, сколько понадобится.

Например, если `условие-2` имеет значение `true`, блок в ❷ будет выполняться и не будет рассматриваться ни один из остальных операторов `if` или `else`. Наконец, блок `else` в ❸ выполняется, если все предыдущие условия вычисляются как `false`. Как и другие блоки `if`, блок `else` является необязательным.

Шаблон функции в листинге 8.14 преобразует аргумент `else` в `Positive`, `Negative` или `Zero`.

Листинг 8.14. Пример использования оператора if

```
#include <cstdio>

template<typename T>
constexpr const char* sign(const T& x) {
  const char* result{};
  if (x == 0) { ❶
    result = "zero";
  } else if (x > 0) { ❷
    result = "positive";
  } else { ❸
    result = "negative";
  }
  return result;
}

int main() {
  printf("float 100 is %s\n", sign(100.0f));
  printf("int -200 is %s\n", sign(-200));
  printf("char 0 is %s\n", sign(char{}));
}
--------------------------------------------------------------------------------
float 100 is positive
int -200 is negative
char 0 is zero
```

Функция `sign` принимает один аргумент и определяет, равен ли он 0 ❶, больше 0 ❷ или меньше 0 ❸. В зависимости от того, какое условие соответствует истине, она устанавливает значение автоматической переменной результата равным одной из трех строк — `zero`, `positive` или `negative` — и возвращает это значение вызывающей стороне.

Операторы инициализации и if

Можно привязать область видимости объекта к оператору `if`, добавив оператор с инициализацией к объявлениям `if` и `elseif`, как показано в листинге 8.15.

Листинг 8.15. Оператор if с инициализацией

```
if (оператор-инициализации; условие-1) {
  // Выполняется, только если условие-1 равно true
} else if (оператор-инициализации; условие-2) { // необязательно
  // Выполняется, только если условие-2 равно true
}
--пропуск--
```

Можно использовать этот шаблон со структурированными привязками для элегантной обработки ошибок. Листинг 8.16 переписывает листинг 8.11 с помощью оператора инициализации для выделения `TextFile` в операторе `if`.

Листинг 8.16. Расширение листинга 8.11 с использованием структурированной привязки и оператора if для обработки ошибок

```
#include <cstdio>

struct TextFile {
  bool success;
  const char* data;
  size_t n_bytes;
};

TextFile read_text_file(const char* path) {
  --пропуск--
}

int main() {
  if(const auto [success, txt, len]❶ = read_text_file("REAMDE.txt"); success❷)
  {
    printf("Read %d bytes: %s\n", len, txt); ❸
  } else {
    printf("Failed to open REAMDE.txt."); ❹
  }
}
--------------------------------------------------------------------------------
Read 27 bytes: Sometimes the goat is you. ❸
```

Объявление структурированной привязки было перемещено в часть оператора инициализации оператора `if` ❶. Это ограничивает каждый из распакованных объектов — `success`, `txt` и `len` — блоком `if`. `success` используется непосредственно в условном выражении `if`, чтобы определить, было ли выполнение `read_text_file` успешным ❷. Если это так, выводится содержимое `REAMDE.txt` ❸. Если это не так, выводится сообщение об ошибке ❹.

Операторы constexpr if

Можно пометить оператор `if` ключевым словом `constexpr`; такие операторы известны как операторы `constexpr if`. `Constexpr if` оператор вычисляется во время компиляции. Блоки кода, соответствующие истинным условиям, выводятся, а остальные игнорируются.

Использование `constexpr if` следует за обычным оператором `if`, как показано в листинге 8.17.

Листинг 8.17. Использование оператора constexpr if

```
if constexpr (условие-1) {
  // Компилируется, только если условие-1 равно true
} else if constexpr (условие-2) { // необязательно; может быть несколько условий
else if
  // Компилируется, только если условие-2 равно true
}
--пропуск--
```

```
} else { // необязательно
  // Компилируется, только если ни одно из условий не равно true
}
```

В сочетании с шаблонами и заголовком `<type_traits>` операторы `constexpr if` являются чрезвычайно мощными. Основное использование `constexpr if` заключается в предоставлении настраиваемого поведения в шаблоне функции в зависимости от некоторых атрибутов параметров типа.

Шаблон функции `value_of` в листинге 8.18 принимает указатели, ссылки и значения. В зависимости от того, какого типа объект является аргументом, `value_of` возвращает либо указанное значение, либо само значение.

Листинг 8.18. Пример шаблона функции value_of, использующий оператор constexpr if

```
#include <cstdio>
#include <stdexcept>
#include <type_traits>

template <typename T>
auto value_of(T x❶) {
  if constexpr (std::is_pointer<T>::value) { ❷
    if (!x) throw std::runtime_error{ "Null pointer dereference." }; ❸
    return *x; ❹
  } else {
    return x; ❺
  }
}

int main() {
  unsigned long level{ 8998 };
  auto level_ptr = &level;
  auto &level_ref = level;
  printf("Power level = %lu\n", value_of(level_ptr)); ❻
  ++*level_ptr;
  printf("Power level = %lu\n", value_of(level_ref)); ❼
  ++level_ref;
  printf("It's over %lu!\n", value_of(level++)); ❽
  try {
    level_ptr = nullptr;
    value_of(level_ptr);
  } catch(const std::exception& e) {
    printf("Exception: %s\n", e.what()); ❾
  }
}
--------------------------------------------------------------------------------
Power level = 8998 ❻
Power level = 8999 ❼
It's over 9000! ❽
Exception: Null pointer dereference. ❾
```

Шаблон функции `value_of` принимает один аргумент `x` ❶. Вы определяете, является ли аргумент типом указателя, используя типаж типа `std::is_pointer<T>` в качестве условного выражения в выражении `constexpr if` ❷. Если `x` является типом указателя, выполняется проверка на `nullptr` и выдается исключение в соответствующем случае ❸. Если `x` не является `nullptr`, он разыменовывается и возвращается результат ❹. В противном случае `x` не является типом указателя, поэтому он возвращается (это, следовательно, значение) ❺.

Внутри `main` создается значение `value_of` несколько раз с указателем `unsigned long` ❻, ссылками `unsigned long` ❼, `unsigned long` ❽ и `nullptr` ❾ соответственно.

Во время выполнения оператор `constexpr if` исчезает; каждое создание значения `value_of` содержит одну ветвь оператора выбора или другую. Возникает вопрос, почему такое средство полезно. В конце концов, программы должны делать что-то полезное во время выполнения, а не во время компиляции. Просто вернитесь к листингу 7.17, и вы увидите, что оценка времени компиляции может существенно упростить программы, исключив магические значения.

Есть и другие примеры, когда оценка времени компиляции популярна, особенно при создании библиотек. Поскольку авторы библиотек обычно не могут знать, как их пользователи будут использовать библиотеку, им нужно написать общий код. Часто они используют методы вроде тех, что вы изучили в главе 6, чтобы достичь полиморфизма во время компиляции. Такие конструкции, как `constexpr`, могут помочь при написании подобного кода.

ПРИМЕЧАНИЕ

Если у вас есть опыт программирования на C, вы сразу узнаете полезность вычислений во время компиляции, ведь она почти полностью заменяет необходимость использования макросов препроцессора.

Операторы switch

Глава 2 впервые представила почтенное выражение `switch`. В этом разделе рассматривается добавление оператора инициализации в объявление `switch`. Использование заключается в следующем:

```
switch (выражение-инициализации❶; условие) {
  case (случай-a): {
    // Обработка случая-a
  } break;
  case (случай-b): {
    // Обработка случая-b
  } break;
    // Обработка других условий при необходимости
  default: {
    // Обработка случая по умолчанию
  }
}
```

Как и в случае с утверждениями `if`, можно создавать экземпляры внутри операторов `switch` ❶. В листинге 8.19 используется оператор инициализации внутри оператора `switch`.

Листинг 8.19. Использование выражения инициализации в операторе switch

```
#include <cstdio>

enum class Color { ❶
  Mauve,
  Pink,
  Russet
};

struct Result { ❷
  const char* name;
  Color color;
};

Result observe_shrub(const char* name) { ❸
  return Result{ name, Color::Russet };
}

int main() {
  const char* description;
  switch (const auto result❹ = observe_shrub("Zaphod"); result.color❺) {
  case Color::Mauve: {
    description = "mauvey shade of pinky russet";
    break;
  } case Color::Pink: {
    description = "pinky shade of mauvey russet";
    break;
  } case Color::Russet: {
    description = "russety shade of pinky mauve";
    break;
  } default: {
    description = "enigmatic shade of whitish black";
  }}
  printf("The other Shaltanac's joopleberry shrub is "
         "always a more %s.", description); ❻
}
--------------------------------------------------------------------------------
The other Shaltanac's joopleberry shrub is always a more russety shade of pinky mauve. ❻
```

Здесь объявляется знакомый класс `Colorenum` ❶ и присоединяется к члену `char*`, чтобы сформировать `Result` ❷ типа POD. Функция `observe_shrub` возвращает `Result` ❸. В `main` вызывается `observe_shrub` в выражении инициализации, а результат сохраняется в переменной `result` ❹. Внутри условного выражения `switch` извлекается элемент `color` этого `result` ❺. Этот элемент определяет случай, который должен выполниться (и устанавливает указатель `description`) ❻.

Как и в случае с синтаксисом, выражение-`if`-плюс-инициализатор, любой объект, инициализированный в выражении инициализации, связан с областью действия оператора `switch`.

Операторы перебора

Операторы перебора (итерации) выполняют сами себя повторно. Четыре вида операторов перебора — это цикл `while`, цикл `do-while`, цикл `for` и цикл `for` на основе диапазона.

Циклы while

Цикл `while` — это основной механизм итерации. Использование его заключается в следующем:

```
while (условие) {
  // Оператор в теле цикла
  // выполняется в течение каждой итерации
}
```

Перед выполнением итерации цикла `while` вычисляет выражение условия. Если значение равно `true`, цикл продолжается. Если оно равно `false`, цикл завершается, как показано в листинге 8.20.

Листинг 8.20. Программа, которая удваивает uint8_t и выводит новое значение на каждой итерации

```
#include <cstdio>
#include <cstdint>

bool double_return_overflow(uint8_t& x) { ❶
  const auto original = x;
  x *= 2;
  return original > x;
}

int main() {
  uint8_t x{ 1 }; ❷
  printf("uint8_t:\n===\n");
  while (!double_return_overflow(x)❸) {
    printf("%u ", x); ❹
  }
}
--------------------------------------------------------------------------------
uint8_t:
===
2 4 8 16 32 64 128 ❹
```

Выше объявлена функция `double_return_overflow`, принимающая 8-разрядное целое беззнаковое число по ссылке ❶. Эта функция удваивает аргумент и проверяет, не вызывает ли это переполнения. Если это так, возвращается `true`. Если переполнения не происходит, возвращается `false`.

Переменная инициализируется значениями от `x` до 1 перед входом в цикл `while` ❷. Условное выражение в цикле `while` вычисляет `double_return_overflow(x)` ❸. У это-

го есть побочный эффект удвоения `x`, потому что он был передан по ссылке. Он также возвращает значение, указывающее, вызвало ли удвоение переполнение `x`. Цикл будет выполняться, если условное выражение вычисляется как `true`, но оно записывается в `double_return_overflow`, поэтому функция возвращает `true`, когда цикл должен остановиться. Эта проблема решается путем добавления оператора логического отрицания (!). (Вспомните из главы 7, что он превращает истину в ложь и ложь в истину.) Таким образом, цикл `while` на самом деле спрашивает: «Если не истинно, что `double_return_overflow` возвращает истинное значение...»

Конечным результатом является вывод значений 2, затем 4, затем 8 и т. д. — до 128 ❹.

Обратите внимание, что значение 1 никогда не выводится, потому что вычисление условного выражения удваивает `x`. Можно изменить это поведение, поместив условный оператор в конец цикла, что приводит к циклу `do-while`.

Циклы do-while

Цикл `do-while` идентичен циклу `while`, за исключением того, что условный оператор вычисляется после завершения цикла, а не до. Его использование заключается в следующем:

```
do {
  // Оператор в теле цикла
  // выполняется при каждой итерации
} while (условие);
```

Поскольку условие вычисляется в конце цикла, гарантируется, что цикл будет выполнен хотя бы один раз.

Листинг 8.21 переписывает листинг 8.20 в цикл `do-while`.

Листинг 8.21. Программа, которая удваивает uint8_t и печатает новое значение на каждой итерации

```
#include <cstdio>
#include <cstdint>

bool double_return_overflow(uint8_t& x) {
  --пропуск--
}

int main() {
  uint8_t x{ 1 };
  printf("uint8_t:\n===\n");
  do {
    printf("%u ", x); ❶
  } while (!double_return_overflow(x)❷);
}
--------------------------------------------------------------------------------
uint8_t:
===
1 2 4 8 16 32 64 128 ❶
```

Обратите внимание, что вывод из листинга 8.21 теперь начинается с 1 ❶. Все, что нужно было сделать, — это переформатировать цикл `while`, чтобы поместить условие в конец цикла ❷.

В большинстве ситуаций, включающих итерации, существуют три задачи:

1. Инициализировать некоторый объект.
2. Обновлять объект перед каждой итерацией.
3. Проверять соответствие объекта некоторому условию.

Можно использовать цикл `while` или `do-while` для выполнения части этих задач, но цикл `for` предоставляет встроенные средства, облегчающие жизнь.

Циклы for

Цикл `for` — это оператор итерации, содержащий три специальных выражения: *инициализация*, *условие* и *итерация*, как описано в следующих разделах.

Выражение инициализации

Выражение инициализации похоже на инициализацию `if`: оно выполняется только один раз перед выполнением первой итерации. Любые объекты, объявленные в выражении инициализации, имеют время жизни, ограниченное областью действия цикла `for`.

Условное выражение

Условное выражение цикла `for` вычисляется непосредственно перед каждой итерацией цикла. Если условное значение равно `true`, цикл продолжает выполняться.

Если условное выражение равно `false`, цикл завершается (это поведение точно такое же, как условное выполнение циклов `while` и `do-while`).

Подобно операторам `if` и `switch`, `for` позволяет инициализировать объекты с областью действия, равной области действия оператора.

Выражение итерации

После каждой итерации цикла `for` вычисляется итерационное выражение. Это происходит до того, как вычисляется условное выражение. Обратите внимание, что итерационное выражение вычисляется после успешной итерации, поэтому итерационное выражение не будет выполнено до первой итерации.

Типичный порядок выполнения в цикле `for` описывает следующий список:

1. Выражение инициализации.
2. Условное выражение.
3. (Тело цикла.)

4. Итерационное выражение.
5. Условное выражение.
6. (Тело цикла.)

Шаги с 4 по 6 повторяются до тех пор, пока условное выражение не вернет `false`.

Применение

Листинг 8.22 демонстрирует использование цикла `for`.

Листинг 8.22. Использование цикла for

```
for(инициализация❶; условное-выражение❷; итерация❸) {
  // Оператор в теле цикла
  // выполняется при каждой итерации
}
```

Выражения инициализации ❶, условного выражения ❷ и итерации ❸ находятся в круглых скобках, предшествующих телу цикла `for`.

Итерация с индексом

Циклы `for` отлично подходят для итерации по составным элементам объекта, подобного массиву. Вспомогательная *индексная* переменная используется для итерации по диапазону допустимых индексов для объекта, подобного массиву. Можно использовать этот индекс для взаимодействия с каждым элементом массива в последовательности. Листинг 8.23 использует индексную переменную для вывода каждого элемента массива вместе с его индексом.

Листинг 8.23. Программа, перебирающая массив чисел Фибоначчи

```
#include <cstdio>

int main() {
  const int x[]{ 1, 1, 2, 3, 5, 8 }; ❶
  printf("i: x[i]\n"); ❷
  for (int i{}❸; i < 6❹; i++❺) {
    printf("%d: %d\n", i, x[i]);
  }
}
--------------------------------------------------------------------------------
i: x[i] ❷
0: 1
1: 1
2: 2
3: 3
4: 5
5: 8
```

Массив `int` инициализируется с именем `x` первыми шестью числами Фибоначчи ❶. После вывода заголовка для выходного значения ❷ создается цикл `for`, содержащий

выражения инициализации ❸, условия ❹ и итерации ❺. Выражение инициализации выполняется первым, и оно инициализирует индексную переменную `i` нулем.

В листинге 8.23 показан шаблон кодирования, который не менялся с 1950-х годов. Можно исключить много шаблонного кода, используя более современный цикл `for` на основе диапазона.

Циклы for на основе диапазонов

Цикл `for` на основе диапазона перебирает диапазон значений без необходимости использования индексной переменной. Диапазон (или выражение диапазона) — это объект, о котором цикл `for` знает, как выполнять итерации по нему. Многие объекты C++ являются допустимыми выражениями диапазона, включая массивы. (Все контейнеры stdlib, о которых вы узнаете во второй части, также являются допустимыми выражениями диапазона.)

Применение

Использование цикла на основе диапазона выглядит следующим образом:

```
for(объявление-диапазона : выражение-диапазона) {
  // Оператор в теле цикла
  // выполняется при каждой итерации
}
```

Объявление диапазона объявляет именованную переменную. Эта переменная должна иметь тот же тип, что и выражение диапазона (можно использовать `auto`).

Листинг 8.24 — это рефакторинг листинга 8.23 для использования цикла `for`, основанного на диапазоне.

Листинг 8.24. Основанный на диапазоне цикл для итерации по первым шести числам Фибоначчи

```
#include <cstdio>

int main() {
  const int x[]{ 1, 1, 2, 3, 5, 8 }; ❶
  for (const auto element❷ : x❸) {
    printf("%d ", element❹);
  }
}
--------------------------------------------------------------------------------
1 1 2 3 5 8
```

По-прежнему объявляется массив `x`, содержащий шесть чисел Фибоначчи ❶. Цикл `for`, основанный на диапазоне, содержит выражение объявления диапазона ❷, в котором объявляется переменная `element` для хранения каждого элемента диапазона. Он также содержит выражение диапазона `x` ❸, в котором хранятся перебираемые и выводимые в консоль элементы ❹.

Код стал намного чище!

Выражения диапазона

Можно определить собственные типы, которые также являются допустимыми выражениями диапазона. Однако нужно указать несколько функций для пользовательского типа.

Каждый диапазон предоставляет методы `begin` и `end`. Эти функции представляют общий интерфейс, который использует цикл на основе диапазона для взаимодействия с диапазоном. Оба метода возвращают *итераторы*. Итератор — это объект, который поддерживает `operator!=`, `operator++` и `operator*`.

Давайте посмотрим, как все эти части сочетаются друг с другом. Под капотом цикл `for` на основе диапазона выглядит так же, как цикл в листинге 8.25.

Листинг 8.25. Цикл for, имитирующий цикл for на основе диапазона

```
const auto e = range.end();❶
for(auto b = range.begin()❷; b != e❸; ++b❹) {
  const auto& element❺ = *b;
}
```

Выражение инициализации хранит две переменные, `b` ❷ и `e` ❶, которые инициализируются в `range.begin()` и `range.end()` соответственно. Условное выражение проверяет, равно ли `b` `e`, и в этом случае цикл завершается ❸ (условно). Выражение итерации увеличивается с помощью префиксного оператора ❹. Наконец, итератор поддерживает оператор разыменования *, поэтому можно извлечь указанный элемент ❺.

ПРИМЕЧАНИЕ

Типы, возвращаемые `begin` и `end`, необязательно должны совпадать. Требование состоит в том, что `operator!=` в `begin` принимает аргумент `end` для поддержки сравнения `begin!= end`.

Диапазон Фибоначчи

Можно реализовать `FibonacciRange`, который будет произвольно генерировать длинную последовательность чисел Фибоначчи. Из предыдущего раздела вы знаете, что этот диапазон должен предлагать методы `begin` и `end`, которые возвращают итератор. Этот итератор, в данном примере называющийся `FibonacciIterator`, должен, в свою очередь, предлагать `operator!=`, `operator++` и `operator*`.

В листинге 8.26 реализованы `FibonacciIterator` и `FibonacciRange`.

Листинг 8.26. Реализация FibonacciIterator и FibonacciRange

```
struct FibonacciIterator {
  bool operator!=(int x) const {
    return x >= current; ❶
  }

  FibonacciIterator& operator++() {
    const auto tmp = current; ❷
```

```
    current += last; ❸
    last = tmp; ❹
    return *this; ❺
  }

  int operator*() const {
    return current; ❻
  }
private:
  int current{ 1 }, last{ 1 };
};

struct FibonacciRange {
  explicit FibonacciRange(int max❼) : max{ max } { }
  FibonacciIterator begin() const { ❽
    return FibonacciIterator{};
  }
  int end() const { ❾
    return max;
  }
private:
  const int max;
};
```

`FibonacciIterator` имеет два поля, `current` и `last`, которые инициализируются значением 1. Они отслеживают два значения в последовательности Фибоначчи. Ее `operator!=` проверяет, является ли аргумент больше или равным `current` ❶. Напомним, что этот аргумент используется в цикле `for` на основе диапазона в условном выражении. Он должен вернуть `true`, если элементы остаются в диапазоне; в противном случае возвращается `false`. `operator++` появляется в выражении итерации и отвечает за настройку итератора для следующей итерации. Сначала сохраняется значение `current` во временную переменную `tmp` ❷. Затем текущее значение увеличивается на значение `last`, получая следующее число Фибоначчи ❸. (Это следует из определения последовательности Фибоначчи.) Затем `last` устанавливается равным `tmp` ❹ и возвращает ссылку на `this` ❺. Наконец, реализуется `operator*`, который возвращает `current` ❻ напрямую.

`FibonacciRange` намного проще. Его конструктор принимает аргумент `max`, который определяет верхний предел для диапазона ❼. Метод `begin` возвращает новый `FibonacciIterator` ❽, а метод `end` возвращает `max` ❾.

Теперь должно быть понятно, почему нужно реализовать `bool operator!= (int x)` в `FibonacciIterator`, а, например, не в `bool operator!=(const FibonacciIterator& x)`: `FibonacciRange` возвращает `int` из `end()`.

Можно использовать `FibonacciRange` в цикле `for` на основе диапазона, как показано в листинге 8.27.

Потребовалась небольшая работа для реализации `FibonacciIterator` и `FibonacciRange` в листинге 8.26, но отдача является существенной. В рамках `main` просто создается

`FibonacciRange` с желаемым верхним пределом ❶, а основанный на диапазоне цикл `for` позаботится обо всем остальном самостоятельно. Вы просто используете полученные элементы внутри цикла `for` ❷.

Листинг 8.27. Использование FibonacciRange в программе

```
#include <cstdio>

struct FibonacciIterator {
  --пропуск--
};

struct FibonacciRange {
  --пропуск--;
};

int main() {
  for (const auto i : FibonacciRange{ 5000 }❶) {
    printf("%d ", i); ❷
  }
}
--------------------------------------------------------------------------------
1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 ❷
```

Листинг 8.27 функционально эквивалентен листингу 8.28, который преобразует основанный на диапазоне цикл `for` в традиционный цикл `for`.

Листинг 8.28. Рефакторинг листинга 8.27, использующий традиционный цикл for

```
#include <cstdio>

struct FibonacciIterator {
  --пропуск--
};

struct FibonacciRange {
  --пропуск--;
};

int main() {
  FibonacciRange range{ 5000 };
  const auto end = range.end(); ❶
  for (auto x = range.begin()❷; x != end ❸; ++x ❹) {
    const auto i = *x;
    printf("%d ", i);
  }
}
--------------------------------------------------------------------------------
1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181
```

Листинг 8.28 демонстрирует, как все части сочетаются друг с другом. Вызов `range.begin()` ❷ возвращает `FibonacciIterator.range.end()` ❶ при вызове возвращает `int`. Эти типы поступают прямо из определений методов `begin()` и `end()`в `FibonacciRange`.

Условный оператор ❸ использует `operator!=(int)` в `FibonacciIterator`, чтобы получить следующее поведение: если итератор `x` прошел через аргумент `int` для `operator!=`, условный оператор принимает значение `false` и цикл завершается. Также реализован `operator++` в `FibonacciIterator`, поэтому `++x` ❹ увеличивает число Фибоначчи в пределах `FibonacciIterator`.

При сравнении листингов 8.27 и 8.28 вы увидите, сколько скуки прячется в циклах `for` на основе диапазона.

ПРИМЕЧАНИЕ

Можно подумать: конечно, основанный на диапазоне цикл `for` выглядит намного чище, но реализация `FibonacciIterator` и `FibonacciRange` — это большая работа. Это замечательный момент, и одноразовый код вы, вероятно, не будете переписывать таким способом. Диапазоны в основном полезны при написании библиотечного кода, который будет часто использоваться, или просто при использовании диапазонов, написанных кем-то другим.

Инструкции перехода

Инструкции перехода, включая операторы `break`, `continue` и `goto`, передают поток управления. В отличие от операторов выбора, операторы перехода не являются условными. Следует избегать их использования, потому что они почти всегда могут быть заменены структурами управления более высокого уровня. Они обсуждаются здесь, потому что их можно увидеть в более старом коде C++ и они все еще играют центральную роль во многих программах на C.

Операторы break

Оператор `break` завершает выполнение включающей итерации или оператора `switch`. После завершения `break` выполнение передается оператору, который следует сразу за оператором `for`, `for` на основе диапазона, `while`, `do-while` или `switch`.

Вы уже использовали `break` в выражениях `switch`; когда случай завершается, оператор `break` завершает `switch`. Напомним, что без оператора `break` оператор `switch` продолжит выполнение всех следующих случаев.

Листинг 8.29 переписывает листинг 8.27, чтобы выйти из цикла `for` на основе диапазона, если итератор `i` равен 21.

Листинг 8.29. Рефакторинг листинга 8.27, который вызывает break, если итератор равен 21

```
#include <cstdio>

struct FibonacciIterator {
  --пропуск--
};
```

```
struct FibonacciRange {
  --пропуск--;
};

int main() {
  for (auto i : FibonacciRange{ 5000 }) {
    if (i == 21) { ❶
      printf("*** "); ❷
      break; ❸
    }
    printf("%d ", i);
  }
}
--------------------------------------------------------------------------------
1 2 3 5 8 13 *** ❷
```

Добавлен оператор `if`, который проверяет, равен ли `i` 21 ❶. Если это так, выводятся три звездочки `***` ❷ и `break`. ❸ Обратите внимание на вывод: вместо вывода 21 программа выводит три звездочки и цикл `for` завершается. Сравните это с выводом из листинга 8.27.

Инструкции continue

Оператор `continue` пропускает оставшуюся часть вмещающего оператора итерации и продолжает следующую итерацию. Листинг 8.30 заменяет `break` в листинге 8.29 на `continue`.

Листинг 8.30. Рефакторинг листинга 8.29 с использованием continue вместо break

```
#include <cstdio>

struct FibonacciIterator {
  --пропуск--
};

struct FibonacciRange {
  --пропуск--;
};

int main() {
  for (auto i : FibonacciRange{ 5000 }) {
    if (i == 21) {
      printf("*** "); ❶
      continue; ❷
    }
    printf("%d ", i);
  }
}
--------------------------------------------------------------------------------
1 2 3 5 8 13 *** ❶34 55 89 144 233 377 610 987 1597 2584 4181
```

По-прежнему выводятся три звездочки ❶, когда `i` равна 21, но вместо `break` используется `continue` ❷. Из-за этого 21 не выводится, как показано в листинге 8.29; однако, в отличие от листинга 8.29, листинг 8.30 продолжает повторяться. (Сравните вывод.)

Инструкции goto

Оператор `goto` — это безусловный переход. Целью оператора `goto` является метка.

Метки

Метки — это идентификаторы, которые можно добавить к любому утверждению. Метки дают операторам названия, и они не имеют прямого влияния на программу. Чтобы назначить метку, перед оператором необходимо указать желаемое имя метки и двоеточие.

Листинг 8.31 добавляет метки `luke` и `yoda` к простой программе.

Листинг 8.31. Простая программа с метками

```
#include <cstdio>

int main() {
luke: ❶
  printf("I'm not afraid.\n");
yoda: ❷
  printf("You will be.");
}
--------------------------------------------------------------------------------
I'm not afraid.
You will be.
```

Метки ❶❷ сами по себе ничего не делают.

Использование goto

Использование оператора `goto` выглядит следующим образом:

```
goto метка;
```

Например, можно использовать операторы `goto`, чтобы излишне запутать простую программу в листинге 8.32.

Листинг 8.32. Код, демонстрирующий оператор goto

```
#include <cstdio>

int main() {
  goto silent_bob; ❶
luke:
  printf("I'm not afraid.\n");
  goto yoda; ❸
silent_bob:
  goto luke; ❷
yoda:
```

```
  printf("You will be.");
}
--------------------------------------------------------------------------------
I'm not afraid.
You will be.
```

Поток управления в листинге 8.32 переходит к `silent_bob` ❶, затем к `luke` ❷, а затем к `yoda` ❸.

Роль goto в современных программах на C++

В современном C++ нет подходящей роли для операторов `goto`. Не стоит их использовать.

ПРИМЕЧАНИЕ

В плохо написанном C++ (и в большинстве кода на C) можно увидеть, что `goto` используется в качестве примитивного механизма обработки ошибок. Во многих случаях системное программирование влечет за собой получение ресурсов, проверку на наличие ошибок и очистку ресурсов. Парадигма RAII аккуратно абстрагирует все эти детали, но у C нет RAII. См. «Увертюру для C-программистов» на с . 36 для получения дополнительной информации.

Итоги

В этой главе вы проработали различные виды операторов, которые можно использовать в программах. Они включали объявления и инициализации, операторы выбора и операторы итерации.

ПРИМЕЧАНИЕ

Имейте в виду, что блоки `try-catch` также являются операторами, но они уже подробно обсуждались в главе 4.

Упражнения

8.1. Перепишите листинг 8.27 на отдельные единицы трансляции: одна для `main` и другая для `FibonacciRange` и `FibonacciIterator`. Используйте заголовочный файл для обмена определениями между двумя единицами трансляции.

8.2. Реализуйте класс `PrimeNumberRange`, который можно использовать в выражении диапазона для перебора всех простых чисел, меньших заданного значения. Опять же используйте отдельный заголовок и исходный файл.

8.3. Интегрируйте `PrimeNumberRange` в листинг 8.27, добавив еще один цикл, который генерирует все простые числа менее 5000.

Что еще почитать?

- Международный стандарт ИСО/МЭК (2017) — Язык программирования C++ (Международная организация по стандартизации; Женева, Швейцария; isocpp.org/std/the-standard/)
- «Random Number Generation and Monte Carlo Methods», 2nd Edition, James E. Gentle (Springer-Verlag, 2003)
- «Random Number Generation and Quasi-Monte Carlo Methods», Harald Niederreiter (SIAM Vol. 63, 1992)

9 Функции

Функции должны делать только что-то одно и очень хорошо.

Роберт С. Мартин, «Чистый код»

Эта глава завершает обсуждение функций, которые инкапсулируют код в повторно используемые компоненты. Теперь, когда вы хорошо разбираетесь в основах C++, в этой главе сначала рассматриваются функции с гораздо более глубоким рассмотрением модификаторов, спецификаторов и возвращаемых типов, которые появляются в объявлениях функций и специализируют поведение ваших функций.

Затем вы узнаете о разрешении перегрузки и принятии переменного числа аргументов, прежде чем исследовать указатели функций, псевдонимы типов, объекты функций и почтенное лямбда-выражение. Глава заканчивается введением в `std::function` перед повторным рассмотрением основной функции и принятием аргументов командной строки.

Объявления функций

Объявления функций имеют следующую знакомую форму:

```
префиксные-модификаторы возвращаемый-тип имя-функции (аргументы)
суффиксные-модификаторы;
```

Можно предоставить ряд необязательных *модификаторов* (или *спецификаторов*) для функций. Модификаторы каким-то образом изменяют поведение функции. Некоторые модификаторы появляются в начале объявления или определения функции (*префиксные модификаторы*), тогда как другие появляются в конце (*суффиксные модификаторы*). Префиксные модификаторы появляются перед возвращаемым типом. Суффиксные — после списка аргументов.

Не существует четкой языковой причины, по которой определенные модификаторы появляются в виде префиксов или суффиксов: поскольку C++ имеет длинную историю, эти функции развивались постепенно.

Префиксные модификаторы

К этому моменту вы уже знаете несколько префиксных модификаторов.

- Префикс `static` указывает, что функция, не являющаяся членом класса, имеет внутреннюю связь, что означает, что функция не будет использоваться за пределами этой единицы трансляции. К сожалению, это ключевое слово выполняет двойную функцию: если оно модифицирует метод (то есть функцию внутри класса), это означает, что функция связана не с созданием экземпляра класса, а с самим классом (см. главу 4).
- Модификатор `virtual` указывает, что метод может быть переопределен дочерним классом. Модификатор `override` указывает компилятору, что дочерний класс намеревается переопределить виртуальную функцию родителя (см. главу 5).
- Модификатор `constexpr` указывает, что функция должна быть выполнена во время компиляции, если это возможно (см. главу 7).
- Модификатор `[[noreturn]]` указывает, что эта функция не возвращает значения (см. главу 8). Напомним, что этот атрибут помогает компилятору оптимизировать код.

Еще один префиксный модификатор — `inline`, который играет роль в руководстве компилятором при оптимизации кода.

На большинстве платформ вызов функции компилируется в серию инструкций, таких как следующие:

1. Поместите аргументы в регистры и в стек вызовов.
2. Вставьте адрес возврата в стек вызовов.
3. Перейдите к вызываемой функции.
4. После завершения функции перейдите к адресу возврата.
5. Очистите стек вызовов.

Эти шаги обычно выполняются очень быстро, и выигрыш в уменьшенном двоичном размере может быть значительным, если использовать функцию во многих местах.

Встраивание функции означает копирование и вставку содержимого функции непосредственно в путь выполнения, что устраняет необходимость в пяти описанных шагах. Это означает, что когда процессор выполняет код, он немедленно выполнит код функции, а не (скромную) церемонию, требуемую для вызова функции. Если вы предпочитаете это незначительное увеличение скорости по сравнению со сравнимой стоимостью в увеличенном двоичном размере, можно использовать ключевое слово `inline`, чтобы указать это компилятору. Ключевое слово `inline` намекает

оптимизатору компилятора, чтобы функция была непосредственно встроенной, а не выполняла вызов функции.

Добавление `inline` к функции не меняет ее поведение; это просто выражение предпочтения компилятору. Стоит убедиться, что при определении встроенной функции вы делаете это во всех единицах трансляции. Также обратите внимание, что современные компиляторы обычно включают встроенные функции там, где это имеет смысл, особенно если функция не используется вне одной единицы трансляции.

Суффиксные модификаторы

На данный момент из книги вы уже знаете два суффиксных модификатора:

- модификатор `noexcept` указывает, что функция *никогда* не вызовет исключение. Это позволяет определенные оптимизации (см. главу 4);
- модификатор `const` указывает, что метод не будет изменять экземпляр своего класса, что позволяет ссылкам на типы `const` вызывать метод (см. главу 4).

В этом разделе рассматриваются еще три суффиксных модификатора: `final`, `override` и `volatile`.

final и override

Последний модификатор указывает, что метод не может быть переопределен дочерним классом. Это фактически противоположность `virtual`. В листинге 9.1 мы пытаемся переопределить метод `final`, что вызывает ошибку компилятора.

Листинг 9.1. Класс, пытающийся переопределить метод final (этот код не компилируется)

```
#include <cstdio>

struct BostonCorbett {
  virtual void shoot() final❶ {
    printf("What a God we have...God avenged Abraham Lincoln");
  }
};

struct BostonCorbettJunior : BostonCorbett {
  void shoot() override❷ { } // Бах! shoot is final.
};

int main() {
  BostonCorbettJunior junior;
}
```

Этот листинг отмечает метод `shoot` модификатором `final` ❶. В `BostonCorbettJunior`, который наследуется от `BostonCorbett`, вы пытаетесь переопределить метод `shoot` ❷. Это вызывает ошибку компилятора.

Также можно применить ключевое слово `final` ко всему классу, не позволяя этому классу полностью стать родителем, как показано в листинге 9.2.

Листинг 9.2. Программа с классом, пытающимся наследовать от класса final. (Этот код не компилируется.)

```
#include <cstdio>

struct BostonCorbett final❶ {
  void shoot() {
    printf("What a God we have...God avenged Abraham Lincoln");
  }
};

struct BostonCorbettJunior : BostonCorbett❷ { }; // Бах!

int main() {
  BostonCorbettJunior junior;
}
```

Класс `BostonCorbett` помечается как `final` ❶, и это вызывает ошибку компилятора при попытке наследовать его в `BostonCorbettJunior` ❷.

ПРИМЕЧАНИЕ

Ни `final`, ни `override` технически не являются ключевыми словами языка — это идентификаторы. В отличие от ключевых слов, идентификаторы приобретают особое значение только при использовании в определенном контексте. Это означает, что можно использовать `final` и `override` в качестве имен символов в других местах программы, что приводит к безумию таких конструкций, как `virtual void final() override`. Постарайтесь этого не делать.

Всякий раз при использовании наследования интерфейса нужно помечать реализацию классов как `final`, потому что модификатор может побудить компилятор выполнить оптимизацию под названием *девиртуализация*. При девиртуализации виртуальных вызовов компилятор устраняет издержки времени выполнения, связанные с виртуальным вызовом.

volatile

Вспомните из главы 7, что значение изменчивого объекта может измениться в любое время, поэтому компилятор должен рассматривать все обращения к изменчивым объектам как видимые побочные эффекты для целей оптимизации. Ключевое слово `volatile` указывает, что метод может быть вызван для изменчивых объектов. Это аналогично тому, как методы `const` могут применяться к постоянным объектам. Вместе эти два ключевых слова определяют *постоянную/изменчивую квалификацию* метода (или иногда *квалификацию cv*), как показано в листинге 9.3.

В этом листинге объявляется метод `apply` для класса `Distillatevolatile` ❶. Также здесь создается `volatileDistillate` под названием `ethanol` в `main` ❷. Поскольку метод `apply` является `volatile`, его все равно можно вызвать ❸ (даже если `ethanol` является `volatile`).

Листинг 9.3. Использование метода volatile

```
#include <cstdio>
struct Distillate {
  int apply() volatile ❶ {
    return ++applications;
  }
private:
  int applications{};
};

int main() {
  volatile ❷ Distillate ethanol;
  printf("%d Tequila\n", ethanol.apply()❸);
  printf("%d Tequila\n", ethanol.apply());
  printf("%d Tequila\n", ethanol.apply());
  printf("Floor!");
}
--------------------------------------------------------------------------------
1 Tequila ❸
2 Tequila
3 Tequila
Floor!
```

Если не отметить `apply` как `volatile` ❶, компилятор выдаст ошибку при попытке вызвать его ❸. Точно так же, как нельзя вызывать не-`const` метод для постоянного объекта, нельзя вызывать не-`volatile` метод для изменчивого объекта. Подумайте, что произойдет, если бы можно было выполнить такую операцию: не-`volatile` метод является кандидатом на все виды оптимизации компилятора по причинам, изложенным в главе 7: многие виды доступа к памяти могут быть оптимизированы без изменения наблюдаемых побочных эффектов в программе.

Как компилятор должен обрабатывать противоречие, возникающее из-за использования `volatile` объекта, который требует, чтобы все обращения к памяти обрабатывались как наблюдаемые побочные эффекты для не-`volatile` метода? Ответ компилятора состоит в том, что он называет это противоречие ошибкой.

Возвращаемые типы auto

Есть два способа объявить возвращаемое значение функции:

- Основной. Возглавьте объявление функции типом ее возврата, как вы делали все это время.
- Вторичный. Пусть компилятор определит правильный тип возвращаемого значения с помощью `auto`.

Как и в случае вывода типа `auto`, компилятор определяет тип возвращаемого значения, фиксируя тип выполнения.

Эту функцию следует использовать разумно. Поскольку определения функций являются документацией, лучше всего указывать конкретные типы возвращаемых данных, когда они доступны.

auto и шаблоны функций

Основной вариант использования для вывода типа `auto` — с шаблонами функций, где тип возвращаемого значения может зависеть (потенциально сложными способами) от параметров шаблона. Его использование заключается в следующем:

```
auto моя-функция(тип-аргумента1 аргумент1, тип-аргумента2 аргумент2, ...) {
  // верните любой тип, а
  // компилятор решит, что скрывается под auto
}
```

Можно расширить синтаксис `auto`-вывода типа возврата, обеспечив тип возврата в качестве суффикса с помощью оператора стрелки ->. Таким образом, можно добавить выражение, которое вычисляет тип возвращаемого значения функции. Его использование заключается в следующем:

```
auto моя-функция(тип-аргумента1 аргумент1, тип-аргумента2 аргумент2, ...) ->
выражение-типа {
  // возврат объекта с типом, соответствующим
  // выражению-типа выше
}
```

Обычно эта строгая форма не используется, но в определенных ситуациях она может быть полезной. Например, эта форма `auto`-вывода типа обычно связана с выражением типа `decltype`. Выражение типа `decltype` возвращает результирующий тип другого выражения. Его использование заключается в следующем:

```
decltype(выражение)
```

Это выражение разрешается в результирующий тип выражения. Например, следующее выражение `decltype` возвращает `int`, поскольку целочисленный литерал 100 имеет этот тип:

```
decltype(100)
```

За пределами общего программирования с шаблонами `decltype` встречается редко.

Можно комбинировать `auto`-вывод типа и `decltype` для документирования типов возврата шаблонов функций. Рассмотрим функцию `add` в листинге 9.4, которая определяет шаблон `add`, складывающий два аргумента.

Функция `add` использует `auto`-вывод типа с выражением типа `decltype` ❶. Каждый раз при создании экземпляра шаблона с двумя типами `X` и `Y` компилятор вычисляет `decltype(X+Y)` и исправляет возвращаемый тип `add`. В рамках `main` предоставляются три экземпляра. Сначала складываются `double` и `int` ❷. Компилятор определяет,

что `decltype(double{100.}+ int{-10})` является `double`, что фиксирует тип возвращаемого значения этого экземпляра `add`. Это, в свою очередь, устанавливает тип `my_double` как `double` ❷. Существуют два других экземпляра: один для `unsigned int` и `int` (что приводит к `unsigned int` ❸), а другой для `char` и `unsigned long long` (что приводит к `unsigned long long` ❹).

Листинг 9.4. Использование decltype и auto-вывода типа

```
#include <cstdio>

template <typename X, typename Y>
auto add(X x, Y y) -> decltype(x + y) { ❶
  return x + y;
}

int main() {
  auto my_double = add(100., -10);
  printf("decltype(double + int) = double; %f\n", my_double); ❷

  auto my_uint = add(100U, -20);
  printf("decltype(uint + int) = uint; %u\n", my_uint); ❸

  auto my_ulonglong = add(char{ 100 }, 54'999'900ull);
  printf("decltype(char + ulonglong) = ulonglong; %llu\n", my_ulonglong); ❹
}
--------------------------------------------------------------------------------
decltype(double + int) = double; 90.000000 ❷
decltype(uint + int) = uint; 80 ❸
decltype(char + ulonglong) = ulonglong; 55000000 ❹
```

Разрешение перегрузки

Разрешение перегрузки — это процесс, который компилятор выполняет при сопоставлении вызова функции с его надлежащей реализацией.

Вспомните из главы 4, что перегрузки функций позволяют указывать функции с одинаковыми именами, но с разными типами и, возможно, с разными аргументами. Компилятор выбирает нужную среди этих перегрузок, сравнивая типы аргументов в вызове функции с типами в каждом объявлении перегрузки. Компилятор выберет лучший из возможных вариантов, а если это невозможно, выдаст ошибку.

Грубо говоря, процесс сопоставления происходит следующим образом:

1. Компилятор будет искать точное совпадение типов.
2. Компилятор попытается использовать целые числа и числа с плавающей точкой для получения подходящей перегрузки (например, `int` для `long` или `float` для `double`).

3. Компилятор попытается сопоставить типы, используя стандартные преобразования, такие как преобразование целочисленного типа в тип с плавающей точкой или приведение указателя на потомка к указателю на родителя.
4. Компилятор будет искать пользовательское преобразование.
5. Компилятор будет искать вариативную функцию.

Вариативные функции

Вариативные функции принимают переменное количество аргументов. Как правило, указывается точное количество аргументов, которые принимает функция путем явного перечисления всех ее параметров. С помощью функции с переменным числом аргументов можно принимать любое количество аргументов. Вариативная функция `printf` является каноническим примером: предоставляются спецификатор формата и произвольное количество параметров. Поскольку `printf` — это вариативная функция, она принимает любое количество параметров.

ПРИМЕЧАНИЕ

Проницательный питонист заметит непосредственную концептуальную взаимосвязь между функциями с переменными числами и `*args/**kwargs`.

Переменные функции объявляются, помещая `...` в качестве последнего параметра в список аргументов функции. Когда вызывается вариативная функция, компилятор сопоставляет предоставленные аргументы с объявленными аргументами. Любые остатки упаковываются в вариативные аргументы, представленные аргументом `...`.

Нельзя извлекать элементы из вариативных аргументов напрямую. Вместо этого вы получаете доступ к отдельным аргументам, используя служебные функции в заголовке `<cstdarg>`.

В табл. 9.1 перечислены эти служебные функции.

Таблица 9.1. Служебные функции в заголовке `<cstdarg>`

Функция	Описание
`va_list`	Используется для объявления локальной переменной, представляющей вариативные аргументы
`va_start`	Разрешает доступ к вариативным аргументам
`va_end`	Используется для завершения итерации по вариативным аргументам
`va_arg`	Используется для перебора каждого элемента в вариативных аргументах
`va_copy`	Создает копию вариативных аргументов

Использование служебных функций немного запутано и лучше всего представлено в связном примере. Рассмотрим вариативную функцию `sum` в листинге 9.5, которая содержит вариативный аргумент.

Листинг 9.5. Функция sum со списком вариативных аргументов

```
#include <cstdio>
#include <cstdint>
#include <cstdarg>

int sum(size_t n, ...❶) {
  va_list args; ❷
  va_start(args, n); ❸
  int result{};
  while (n--) {
    auto next_element = va_arg(args, int); ❹
      result += next_element;
  }
  va_end(args); ❺
  return result;
}

int main() {
  printf("The answer is %d.", sum(6, 2, 4, 6, 8, 10, 12)); ❻
}
--------------------------------------------------------------------------------
The answer is 42. ❻
```

`Sum` объявляется как вариативная функция ❶. Все вариативные функции должны объявить `va_list`. Здесь он называется `args` ❷. `va_list`требует инициализации с помощью метода `va_start` ❸, который принимает два аргумента. Первый параметр — это `va_list`, а второй — размер вариативных аргументов. Итерация по каждому элементу вариативных аргументов осуществляется при помощи функции `va_args`. Первый ее параметр — это аргумент `va_list`, а второй — тип аргумента ❹. После завершения итерации вызывается `va_end` со структурой `va_list` ❺.

`Sum` вызывается с семью аргументами: первый — количество вариативных аргументов (шесть), за которыми следуют шесть чисел (2, 4, 6, 8, 10, 12) ❻.

Вариативные функции являются пережитком C. Обычно вариативные функции небезопасны и являются общим источником уязвимостей в безопасности.

Существуют по крайней мере две основные проблемы вариативных функций:

- вариативные аргументы не обеспечивают безопасность типов (обратите внимание, что второй аргумент `va_args` является типом);
- количество элементов в вариативных аргументах должно отслеживаться отдельно.

Компилятор не может помочь разработчику ни с одной из этих проблем.

К счастью, вариативные шаблоны обеспечивают более безопасный и эффективный способ реализации вариативных функций.

Вариативные шаблоны

Вариативный шаблон позволяет создавать шаблоны функций, которые принимают аргументы с одинаковым типом переменных. Они позволяют вам использовать значительную мощность шаблонного движка. Чтобы объявить шаблон переменной, добавьте специальный параметр шаблона, называемый *пакетом параметров шаблона*. В листинге 9.6 демонстрируется его использование.

Листинг 9.6. Функция шаблона с пакетом параметров

```
template <typename...❶ Тип-аргументов>
возвращаемый-тип имя-функции(Тип-аргументов ...❷ аргументы) {
  // Используйте семантику пакета параметров
  // в теле функции
}
```

Пакет параметров шаблона является частью списка параметров шаблона ❶. При использовании `Args` в шаблоне функции ❷ он называется пакетом параметров функции. Некоторые специальные операторы доступны для использования с *пакетами параметров*:

- можно использовать `sizeof...(args)` для получения размера пакета параметров;
- можно вызвать функцию (например, `other_function`) со специальным синтаксисом `other_function(args...)`. Это расширяет аргументы пакета параметров и позволяет выполнять дальнейшую обработку аргументов, содержащихся в пакете параметров.

Программирование с помощью блоков параметров

К сожалению, невозможно напрямую проиндексировать пакет параметров. Нужно вызвать шаблон функции изнутри самой себя — процесс, называемый *рекурсией во время компиляции*, — для рекурсивной итерации по элементам в пакете параметров.

В листинге 9.7 приведен паттерн подобного поведения.

Листинг 9.7. Функция шаблона, где показана рекурсия во время компиляции с пакетами параметров

```
template <typename T, typename...Args>
void my_func(T x❶, Args...args) {
  // Используйте x, а затем рекурсию:
  my_func(args...); ❷
}
```

В отличие от других примеров использования, замещающие знаки, содержащиеся в этом листинге, являются литералами.

Суть заключается в добавлении обычного параметра шаблона перед пакетом параметров ❶. Каждый раз при вызове `my_func x` принимает первый аргумент. Остальная

часть упаковывается в `args`. Для вызова используется конструкция `args...`, чтобы развернуть пакет параметров ❷.

Рекурсии нужны критерии остановки, поэтому добавьте специализацию шаблона функции без параметра:

```
template <typename T>
void my_func(T x) {
  // Используйте x, но БЕЗ рекурсии
}
```

Пересмотр функции sum

Рассмотрим (значительно улучшенную) функцию `sum`, реализованную как вариативный шаблон в листинге 9.8.

Листинг 9.8. Рефакторинг листинга 9.5, использующий пакет параметров шаблона вместо va_args

```
#include <cstdio>

template <typename T>
constexpr❶ T sum(T x) { ❷
    return x;
}

template <typename T, typename... Args>
constexpr❸ T sum(T x, Args... args) { ❹
    return x + sum(args...❺);
}

int main() {
  printf("The answer is %d.", sum(2, 4, 6, 8, 10, 12)); ❻
}
--------------------------------------------------------------------------------
The answer is 42. ❻
```

Первая функция ❷ — это перегрузка, которая обрабатывает условие остановки; если функция имеет только один аргумент, просто верните аргумент `x`, потому что сумма одного элемента — это сам элемент. Вариативный шаблон ❹ следует шаблону рекурсии, приведенному в листинге 9.7. Он достает один аргумент `x` из аргументов пакета параметров, а затем возвращает `x` плюс результат рекурсивного вызова `sum` с расширенным пакетом параметров ❺. Поскольку все это общее программирование может быть вычислено во время компиляции, эти функции отмечаются как `constexpr` ❶❸. Это вычисление во время компиляции является *основным* преимуществом по сравнению с листингом 9.5, который имеет идентичный вывод, но вычисляет результат во время выполнения ❻. (Зачем платить за время выполнения, когда это не нужно?)

Если необходимо просто применить один бинарный оператор (например, плюс или минус) к диапазону значений (как в листинге 9.5), можно использовать выражение свертки вместо рекурсии.

Выражения свертки

Выражение свертки вычисляет результат использования бинарного оператора для всех аргументов пакета параметров. Выражения свертки отличны от вариативных шаблонов, но связаны с ними. Их использование заключается в следующем:

```
(... бинарный-оператор пакет-параметров)
```

Например, можно использовать следующее выражение свертки для суммирования всех элементов в пакете параметров с именем `args`:

```
(... + args)
```

Листинг 9.9 является рефакторингом листинга 9.8 и использует выражение свертки вместо рекурсии.

Листинг 9.9. Рефакторинг листинга 9.8 с использованием выражения свертки

```
#include <cstdio>

template <typename... T>
constexpr auto sum(T... args) {
  return (... + args); ❶
}

int main() {
  printf("The answer is %d.", sum(2, 4, 6, 8, 10, 12)); ❷
}
--------------------------------------------------------------------------------
The answer is 42. ❷
```

Функция `sum` упрощается при использовании выражения свертки вместо рекурсивного подхода ❶. Конечные результаты идентичны ❷.

Указатели функций

Функциональное программирование — это парадигма программирования, в которой особое внимание уделяется вычислению функций и неизменным данным. Одной из основных концепций в функциональном программировании является передача функции в качестве параметра другой функции.

Один из способов добиться этого — передать указатель на функцию. Функции занимают память, как и объекты. Можно обратиться к этому адресу в памяти с помощью обычных механизмов указателя. Однако, в отличие от объектов, нельзя изменить указанную функцию. В этом отношении функции концептуально похожи на объекты `const`. Можно взять адрес функций и вызвать их, и это все.

Объявление указателя функции

Чтобы объявить указатель на функцию, используйте следующий уродливый синтаксис:

```
возвращаемый-тип (*имя-указателя)( тип-аргумента1, тип-аргумента2, ...);
```

Это выглядит так же, как объявление функции, где имя функции заменяется на `(*pointer-name)`.

Как обычно, можно использовать оператор вычисления адреса `&`, чтобы получить адрес функции. Однако это необязательно; можно просто использовать имя функции в качестве указателя.

В листинге 9.10 показано, как можно получить и использовать указатели на функции.

Листинг 9.10. Программа, где используются указатели функций. (Из-за случайного распределения адресного пространства адреса ❹❼ будут меняться во время выполнения.)

```
#include <cstdio>

float add(float a, int b) {
  return a + b;
}

float subtract(float a, int b) {
  return a - b;
}

int main() {
  const float first{ 100 };
  const int second{ 20 };

  float(*operation)(float, int) {}; ❶
  printf("operation initialized to 0x%p\n", operation); ❷
  operation = &add; ❸
  printf("&add = 0x%p\n", operation); ❹
  printf("%g + %d = %g\n", first, second, operation(first, second)); ❺

  operation = subtract; ❻
  printf("&subtract = 0x%p\n", operation); ❼
  printf("%g - %d = %g\n", first, second, operation(first, second)); ❽
}
--------------------------------------------------------------------------------
operation initialized to 0x0000000000000000 ❷
&add = 0x00007FF6CDFE1070 ❹
100 + 20 = 120 ❺
&subtract = 0x00007FF6CDFE10A0 ❼
100 - 20 = 80 ❽
```

В этом листинге показаны две функции с одинаковыми сигнатурами: `add` и `subtract`. Поскольку сигнатуры функций совпадают, типы указателей на эти функции также

будут совпадать. Указатель на функцию инициализируется, принимая аргументы с плавающей точкой и `int` в качестве аргументов и возвращая `float` ❶. Затем выводится значение операции, которое является `nullptr`, после инициализации ❷.

Затем адрес `add` назначается `operation` ❸ с помощью оператора вычисления адреса и выводится его новый адрес ❹. Запускается `operation` и выводится результат ❺.

Чтобы показать, что можно переназначить указатели функций, `operation` назначается `subtract` без использования оператора вычисления адреса ❻, выводится новое значение `operation` ❼ и, наконец, выводится результат ❽.

Совмещение имен типов и указатели функций

Псевдонимы типов обеспечивают удобный способ программирования с помощью указателей функций. Использование заключается в следующем:

```
using имя-псевдонима = возвращаемый-тип(*)(тип-аргумента1, тип-аргумента2, ...)
```

Можно было определить псевдоним типа `operation_func` в листинге 9.10, например:

```
using operation_func = float(*)(float, int);
```

Это особенно полезно, если вы будете использовать указатели функций одного типа; это может действительно очистить код.

Оператор вызова функции

Вы можете сделать определяемые пользователем типы вызываемыми (callable) или invocable, перегружая оператор вызова функции `operator()()`. Такой тип называется типом функции, а экземпляры типа функции называются объектами функции. Оператор вызова функции допускает любую комбинацию типов аргументов, возвращаемых типов и модификаторов (кроме `static`).

Основная причина, по которой можно захотеть сделать пользовательский тип вызываемым, заключается во взаимодействии с кодом, который ожидает, что функциональные объекты будут использовать оператор вызова функции. Вы обнаружите, что многие библиотеки, такие как stdlib, используют оператор вызова функции в качестве интерфейса для объектов, подобных функциям. Например, в главе 19 вы узнаете, как создать асинхронную задачу с помощью функции `std::async`, которая принимает произвольный объект функции, который может выполняться в отдельном потоке. Она использует оператор вызова функции в качестве интерфейса. Комитет, который изобрел `std::async`, мог потребовать, чтобы вы выставили, скажем, метод `run`, но они выбрали оператор вызова функции, потому что он позволяет универсальному коду использовать идентичные обозначения для вызова функции или объекта функции.

В листинге 9.11 показано использование оператора вызова функции.

Листинг 9.11. Использование оператора вызова функции

```
struct имя-типа {
  возвращаемый-тип❶ operator()❷(тип-аргумента1 аргумент1, тип-аргумента2
аргумент2, ...❸) {
    // Тело оператора вызова функции
  }
}
```

Оператор вызова функции имеет специальное имя метода `operator()` ❷. Объявляется произвольное количество аргументов ❸, а также выбирается соответствующий тип возврата ❶.

Когда компилятор вычисляет выражение вызова функции, он вызывает оператор вызова функции для первого операнда, передавая оставшиеся операнды в качестве аргументов. Результатом вычисления выражения вызова функции является результат вызова соответствующего оператора вызова функции.

Пример подсчета

Рассмотрим тип функции `CountIf` в листинге 9.12, который вычисляет частоту появления конкретного символа в строке с нулевым символом в конце.

Листинг 9.12. Тип функции, подсчитывающий количество символов в строке с нулевым символом в конце

```
#include <cstdio>
#include <cstdint>

struct CountIf {
  CountIf(char x) : x{ x } { }❶
  size_t operator()(const char* str❷) const {
    size_t index{}❸, result{};
    while (str[index]) {
      if (str[index] == x) result++; ❹
      index++;
    }
    return result;
  }
private:
  const char x;
};

int main() {
  CountIf s_counter{ 's' }; ❺
  auto sally = s_counter("Sally sells seashells by the seashore."); ❻
  printf("Sally: %zu\n", sally);
  auto sailor = s_counter("Sailor went to sea to see what he could see.");
  printf("Sailor: %zu\n", sailor);
  auto buffalo = CountIf{ 'f' }("Buffalo buffalo Buffalo buffalo "
                                "buffalo buffalo Buffalo buffalo."); ❼
```

```
  printf("Buffalo: %zu\n", buffalo);
}
--------------------------------------------------------------------------------
Sally: 7
Sailor: 3
Buffalo: 16
```

Объекты `CountIf` инициализируются с помощью конструктора, принимающего `char` ❶. Можно вызвать получившийся объект функции, как если бы он был функцией, принимающей строку с нулевым символом в конце ❷, потому что реализован оператор вызова функции. Оператор вызова функции выполняет итерацию по каждому символу в аргументе `str`, используя индексную переменную ❸, увеличивая результирующую переменную всякий раз, когда символ соответствует полю `x` ❹. Поскольку вызов функции не изменяет состояние объекта `CountIf`, он помечается как `const`.

В `main` был инициализирован объект функции `CountIf s_counter`, который будет считать частоту буквы `s` ❺. Можно использовать `s_counter`, как если бы это была функция ❻. Вы даже можете инициализировать объект `CountIf` и использовать оператор функции непосредственно как объект с `r`-значением ❼. Это может оказаться удобным в некоторых ситуациях, когда, например, может потребоваться вызвать объект только один раз.

Можно использовать функциональные объекты как частичные приложения. Листинг 9.12 концептуально похож на функцию `count_if` в листинге 9.13.

Листинг 9.13. Свободная функция, эмулирующая листинг 9.12

```
#include <cstdio>
#include <cstdint>

size_t count_if(char x❶, const char* str) {
  size_t index{}, result{};
  while (str[index]) {
    if (str[index] == x) result++;
    index++;
  }
  return result;
}

int main() {
  auto sally = count_if('s', "Sally sells seashells by the seashore.");
  printf("Sally: %zu\n", sally);
  auto sailor = count_if('s', "Sailor went to sea to see what he could see.");
  printf("Sailor: %zu\n", sailor);
  auto buffalo = count_if('f', "Buffalo buffalo Buffalo buffalo "
                               "buffalo buffalo Buffalo buffalo.");
  printf("Buffalo: %zu\n", buffalo);
}
--------------------------------------------------------------------------------
Sally: 7
Sailor: 3
Buffalo: 16
```

Функция `count_if` имеет дополнительный аргумент `x` ❶, но в остальном она почти идентична оператору функции `CountIf`.

ПРИМЕЧАНИЕ

На языке функционального программирования `CountIf` — частичное применение `x` к `count_if`. Когда вы частично применяете аргумент к функции, вы фиксируете значение этого аргумента. Результатом такого частичного применения является другая функция, принимающая на один аргумент меньше.

Объявление типов функций является многословным. Часто можно существенно уменьшить шаблон с помощью лямбда-выражений.

Лямбда-выражения

Лямбда-выражения лаконично создают объекты безымянных функций. Объект функции подразумевает тип функции, что позволяет быстро объявить объект функции. Лямбды не предоставляют никаких дополнительных функций, кроме объявления типов функций старомодным способом. Но они чрезвычайно удобны, когда нужно инициализировать функциональный объект только в одном контексте.

Использование

Лямбда-выражение состоит из пяти компонентов:

- *захват* (captures): переменные-члены объекта функции (то есть частично примененные параметры);
- *параметры* (parameters): параметры, необходимые для вызова объекта функции;
- *тело* (body): код объекта функции;
- *спецификаторы* (specifiers): элементы вроде `constexpr`, `mutable`, `noexcept` и `[[noreturn]]`;
- *тип возвращаемого значения* (return type): тип, возвращаемый объектом функции.

Использование лямбда-выражения выглядит следующим образом:

```
[захват❶] (параметры❷) спецификаторы❺ -> возвращаемый-тип❹ { тело❸ }
```

Обязательны только захват и тело; все остальное необязательно. Подробнее о каждом из этих компонентов вы узнаете в следующих нескольких разделах.

Каждый лямбда-компонент имеет прямой аналог в функциональном объекте. Чтобы сформировать мост между объектами функций, такими как `CountIf`, и лямбда-выражениями, взгляните на листинг 9.14, в котором перечислены типы функций

`CountIf` из листинга 9.12 с аннотациями, которые соответствуют аналогичным частям лямбда-выражения в листинге с примером использования.

Листинг 9.14. Сравнение объявления типа CountIf с лямбда-выражением

```
struct CountIf {
  CountIf(char x) : x{ x } { } ❶
  size_t❹ operator()(const char* str❷) const❺ {
    --пропуск--❸
  }
private:
  const char x; ❷
};
```

Переменные-члены, которые устанавливаются в конструкторе `CountIf`, аналогичны лямбда-захвату ❶. Аргументы оператора вызова функции ❷, тело ❸ и возвращаемый тип ❹ аналогичны параметрам лямбды, телу и возвращаемому типу. Наконец, модификаторы могут применяться к оператору вызова функции ❺ и лямбда-выражению. (Числа в примере использования лямбда-выражения и в листинге 9.14 соответствуют друг другу.)

Тела и параметры лямбда-выражений

Лямбда-выражения производят функциональные объекты. Как функциональные объекты лямбда-выражения могут быть вызваны. В большинстве случаев нужно, чтобы функциональный объект принимал параметры при вызове.

Тело лямбда-выражения похоже на тело функции: все параметры имеют область действия.

Лямбда-параметры и тела объявляются, по существу, с тем же синтаксисом, который используется для функций.

Например, следующее лямбда-выражение возвращает объект функции, который возводит в квадрат свой аргумент `int`:

```
[](int x) { return x*x; }
```

Лямбда-выражение принимает один `int x` и использует его в теле для возведения в квадрат.

В листинге 9.15 используются три разных лямбда-выражения для преобразования массива `1, 2, 3`.

Листинг 9.15. Три лямбда-выражения и функция transform

```
#include <cstdio>
#include <cstdint>

template <typename Fn>
void transform(Fn fn, const int* in, int* out, size_t length) { ❶
  for(size_t i{}; i<length; i++) {
```

```
    out[i] = fn(in[i]); ❷
  }
}

int main() {
  const size_t len{ 3 };
  int base[]{ 1, 2, 3 }, a[len], b[len], c[len];
  transform([](int x) { return 1; }❸, base, a, len);
  transform([](int x) { return x; }❹, base, b, len);
  transform([](int x) { return 10*x+5; }❺, base, c, len);
  for (size_t i{}; i < len; i++) {
    printf("Element %zu: %d %d %d\n", i, a[i], b[i], c[i]);
  }
}
--------------------------------------------------------------------------------
Element 0: 1 1 15
Element 1: 1 2 25
Element 2: 1 3 35
```

Функция шаблона `transform` ❶ принимает четыре аргумента: объект функции `fn`, массивы `in` и `out` и соответствующий размер этих массивов. В рамках преобразования вызывается `fn` для каждого элемента `in`, и результат присваивается соответствующему элементу `out` ❷.

Внутри `main` объявляется базовый массив 1, 2, 3, который будет использоваться как массив `in`. В той же строке также объявляются три неинициализированных массива `a`, `b` и `c`, которые будут использоваться в качестве выходных массивов. Первый вызов `transform` передает лямбда-выражение `([](int x) {return 1;}),` которое всегда возвращает 1 ❸, и результат сохраняется в `a`. (Заметьте, что лямбда-выражению не нужно имя!) Второй вызов `transform([](int x) {return x;})` просто возвращает свой аргумент ❹, и результат сохраняется в `b`. Третий вызов преобразования умножает аргумент на 10 и добавляет 5 ❺. Результат сохраняется в `c`. Затем выходные данные выводятся в матрицу, где каждый столбец демонстрирует преобразование, которое было применено к различным лямбда-выражениям в каждом случае.

Обратите внимание, что `transform` объявлен как функция шаблона, что позволяет повторно использовать его с любым объектом функции.

Параметры по умолчанию

Можно предоставить лямбда-аргументы по умолчанию. Лямбда-аргументы по умолчанию ведут себя так же, как параметры функции по умолчанию. Вызывающая сторона может указывать значения для параметров по умолчанию, и в этом случае лямбда-выражение использует значения, предоставленные вызывающей стороной. Если вызывающая сторона не указывает значение, лямбда-выражение использует значение по умолчанию.

В листинге 9.16 показано поведение аргумента по умолчанию.

Листинг 9.16. Использование лямбда-параметров по умолчанию

```
#include <cstdio>

int main() {
  auto increment = [](auto x, int y = 1❶) { return x + y; };
  printf("increment(10) = %d\n", increment(10)); ❷
  printf("increment(10, 5) = %d\n", increment(10, 5)); ❸
}
-------------------------------------------------------------------------------
increment(10) = 11 ❷
increment(10, 5) = 15 ❸
```

Лямбда-выражение `increment` имеет два параметра: `x` и `y`. Однако параметр `y` является необязательным, поскольку имеет аргумент по умолчанию 1 ❶. Если не был указан аргумент для `y` при вызове функции ❷, `increment` возвращает `1 + x`. Если функция вызывается с аргументом для `y` ❸, вместо этого используется это значение.

Обобщенные лямбда-выражения

Обобщенные лямбда-выражения — это шаблоны лямбда-выражений. Для одного или нескольких параметров вместо конкретного типа указывается `auto`. Эти `auto`-типы становятся параметрами шаблона, что означает, что компилятор исключит пользовательское создание лямбда-выражения.

В листинге 9.17 показано, как назначить обобщенное лямбда-выражение переменной, а затем использовать лямбда-выражение в двух разных экземплярах шаблона.

Листинг 9.17. Использование обобщенного лямбда-выражения

```
#include <cstdio>
#include <cstdint>

template <typename Fn, typename T❶>
void transform(Fn fn, const T* in, T* out, size_t len) {
  for(size_t i{}; i<len; i++) {
    out[i] = fn(in[i]);
  }
}

int main() {
  constexpr size_t len{ 3 };
  int base_int[]{ 1, 2, 3 }, a[len]; ❷
  float base_float[]{ 10.f, 20.f, 30.f }, b[len]; ❸
  auto translate = [](auto x) { return 10 * x + 5; }; ❹
  transform(translate, base_int, a, len); ❺
  transform(translate, base_float, b, len); ❻

  for (size_t i{}; i < l; i++) {
    printf("Element %zu: %d %f\n", i, a[i], b[i]);
  }
}
-------------------------------------------------------------------------------
```

```
Element 0: 15 105.000000
Element 1: 25 205.000000
Element 2: 35 305.000000
```

Второй параметр шаблона добавляется в `transform` ❶, который используется в качестве указателя на `in` и `out`. Это позволяет применять преобразование к массивам любого типа, а не только к типам `int`. Чтобы протестировать обновленный шаблон преобразования, объявляются два массива с разными указанными типами: `int` ❷ и `float` ❸. (Вспомните из главы 3, что `f` в `10.f` определяет литерал с плавающей точкой.) Затем назначается универсальное лямбда-выражение переменной `translate` ❹. Это позволяет использовать одно и то же лямбда-выражение для каждого экземпляра преобразования: при создании экземпляров с `base_int` ❺ и `base_float` ❻.

Без обобщенного лямбда-выражения придется явно объявлять типы параметров, как показано ниже:

```
--пропуск--
  transform([](int x) { return 10 * x + 5; }, base_int, a, l); ❺
  transform([](double x) { return 10 * x + 5; }, base_float, b, l); ❻
```

До сих пор вы полагались на компилятор, чтобы определить типы возврата лямбда-выражений. Это особенно полезно для универсальных лямбда-выражений, поскольку часто тип возвращаемого значения зависит от типов параметров. Но можно явно указать тип возвращаемого значения, если необходимо.

Возвращаемые типы лямбда-выражений

Компилятор определяет возвращаемый тип лямбда-выражения самостоятельно. Чтобы отойти от компилятора, используется синтаксис стрелки ->, как показано ниже:

```
[](int x, double y) -> double { return x + y; }
```

Это лямбда-выражение принимает `int` и `double` и возвращает `double`.

Также можно использовать выражения `decltype`, которые могут быть полезны в случае с обобщенными лямбда-выражениями. Например, рассмотрим следующее лямбда-выражение:

```
[](auto x, double y) -> decltype(x+y) { return x + y; }
```

Здесь объявлено, что типом возврата лямбда-выражения является любой тип, полученный в результате добавления `x` к `y`.

Тип возвращаемого значения лямбда-выражения, как правило, редко указывается явно.

Гораздо более распространенным требованием является то, что нужно добавить объект в лямбда-выражение перед вызовом. Эту роль берет на себя лямбда-захват.

Захват лямбда-выражений

Захват лямбда-выражений вводит объекты в лямбда-выражение. Введенные объекты помогают изменить поведение лямбда-выражения.

Объявите лямбда-захват, указав список захвата в скобках [].

Список захвата идет перед списком параметров и может содержать любое количество разделенных запятыми аргументов. Затем эти аргументы используются в теле лямбда-выражения.

Лямбда-выражение может захватывать аргументы по ссылке или по значению. По умолчанию лямбда-выражения захватывают аргументы по значению.

Список захвата лямбды аналогичен конструктору типа функции.

Листинг 9.18 переформулирует `CountIf` из листинга 9.12 как лямбда-выражение `s_counter`.

Листинг 9.18. Переформулировка CountIf из листинга 9.12 в виде лямбда-выражения

```
#include <cstdio>
#include <cstdint>

int main() {
  char to_count{ 's' }; ❶
  auto s_counter = [to_count❷](const char* str) {
    size_t index{}, result{};
    while (str[index]) {
      if (str[index] == to_count❸) result++;
      index++;
    }
    return result;
  };
  auto sally = s_counter("Sally sells seashells by the seashore."❹);
  printf("Sally: %zu\n", sally);
  auto sailor = s_counter("Sailor went to sea to see what he could see.");
  printf("Sailor: %zu\n", sailor);
}
--------------------------------------------------------------------------------
Sally: 7
Sailor: 3
```

Здесь `char` с именем `to_count` инициализируется буквой `s` ❶. Затем `to_count` фиксируется в лямбда-выражении, присвоенном `s_counter` ❷. Это делает `to_count` доступным в теле лямбда-выражения ❸.

Чтобы захватить элемент по ссылке, а не по значению, перед именем захваченного объекта добавьте амперсанд `&`. В листинге 9.19 добавлена ссылка на захват `s_counter`, которая поддерживает подсчет вызовов лямбда-выражения.

Листинг 9.19. Использование ссылки на захват в лямбде

```
#include <cstdio>
#include <cstdint>

int main() {
  char to_count{ 's' };
  size_t tally{};❶
  auto s_counter = [to_count, &tally❷](const char* str) {
    size_t index{}, result{};
    while (str[index]) {
      if (str[index] == to_count) result++;
      index++;
    }
    tally += result;❸
    return result;
  };
  printf("Tally: %zu\n", tally); ❹
  auto sally = s_counter("Sally sells seashells by the seashore.");
  printf("Sally: %zu\n", sally);
  printf("Tally: %zu\n", tally); ❺
  auto sailor = s_counter("Sailor went to sea to see what he could see.");
  printf("Sailor: %zu\n", sailor);
  printf("Tally: %zu\n", tally); ❻
}
--------------------------------------------------------------------------------
Tally: 0 ❹
Sally: 7
Tally: 7 ❺
Sailor: 3
Tally: 10 ❻
```

Переменная счетчика `tally` инициализируется нулем ❶, а затем лямбда-выражение `s_counter` захватывает `tally` по ссылке (обратите внимание на амперсанд `&`) ❷. В теле лямбды добавляется оператор для приращения счетчика по вызову `result` перед возвратом ❸. В результате `tally` будет отслеживать общее количество независимо от того, сколько раз было вызвано лямбда-выражение. Перед первым вызовом `s_counter` выводится значение `tally` ❹ (которое по-прежнему равно нулю). После вызова `s_counter` со строкой `Sally sells seashells by the seashore.` счет будет равен 7 ❺. Последний вызов`s_counter` `Sailor went to sea to see what he could see.` возвращает 3, поэтому значение `tally` равно 7 + 3 = 10 ❻.

Захват по умолчанию

До сих пор вам приходилось фиксировать каждый элемент по имени. Иногда этот стиль захвата называется *именованным захватом*. При желании можно захватить все автоматические переменные, используемые в лямбда-выражениях, используя *захват по умолчанию*. Чтобы указать захват по умолчанию по значению в списке захвата, используйте одиночный знак равенства =. Чтобы указать захват по умолчанию по ссылке, используйте одиночный амперсанд &.

Например, можно было бы упростить лямбда-выражение в листинге 9.19, чтобы выполнить захват по умолчанию по ссылке, как показано в листинге 9.20.

Листинг 9.20. Упрощение лямбда-выражения с захватом по умолчанию по ссылке

```
--пропуск--
  auto s_counter = [&❶](const char* str) {
    size_t index{}, result{};
    while (str[index]) {
      if (str[index] == to_count❷) result++;
      index++;
    }
    tally❸ += result;
    return result;
  };
--пропуск--
```

Вы указываете захват по умолчанию по ссылке ❶, что означает, что любые автоматические переменные в теле лямбда-выражения захватываются по ссылке. Их два: `to_count` ❷ и `tally` ❸.

Если скомпилировать и запустить переделанный листинг, вы получите идентичный вывод. Однако обратите внимание, что `to_count` теперь захвачен по ссылке. Если случайно изменить его в теле лямбда-выражения, это будет происходить как по лямбда-вызовам, так и внутри `main` (где `to_count` — автоматическая переменная).

Что произойдет, если вместо этого выполнить захват по умолчанию? Нужно всего лишь изменить `=` на `&` в списке захвата, как показано в листинге 9.21.

Листинг 9.21. Изменение листинга 9.20 для захвата по значению, а не по ссылке (этот код не компилируется)

```
--пропуск--
  auto s_counter = [=❶](const char* str) {
    size_t index{}, result{};
    while (str[index]) {
      if (str[index] == to_count❷) result++;
      index++;
    }
    tally❸ += result;
    return result;
  };
--пропуск--
```

Захват по умолчанию меняется на захват по значению ❶. Захват `to_count` не затрагивается ❷, но попытка изменить `tally` приводит к ошибке компилятора ❸. Нельзя изменять переменные, захваченные по значению без добавления ключевого слова `mutable` в лямбда-выражение. Ключевое слово `mutable` позволяет изменять переменные, захваченные по значению. Это включает в себя вызов не-`const` методов для этого объекта.

В листинге 9.22 добавлен модификатор `mutable` и по умолчанию используется захват по значению.

Листинг 9.22. Изменяемое лямбда-выражение с захватом по умолчанию

```
#include <cstdio>
#include <cstdint>

int main() {
  char to_count{ 's' };
  size_t tally{};
  auto s_counter = [=❶](const char* str) mutable❷ {
    size_t index{}, result{};
    while (str[index]) {
      if (str[index] == to_count) result++;
      index++;
    }
    tally += result;
    return result;
  };
  auto sally = s_counter("Sally sells seashells by the seashore.");
  printf("Tally: %zu\n", tally); ❸
  printf("Sally: %zu\n", sally);
  printf("Tally: %zu\n", tally); ❹
  auto sailor = s_counter("Sailor went to sea to see what he could see.");
  printf("Sailor: %zu\n", sailor);
  printf("Tally: %zu\n", tally); ❺
}
--------------------------------------------------------------------------------
Tally: 0
Sally: 7
Tally: 0
Sailor: 3
Tally: 0
```

Захват по умолчанию объявляется захватом по значению ❶, и переменная `s_counter` лямбда-выражения помечается как `mutable` ❷. Каждый из трех раз при выводе `tally` ❸❹❺ получается нулевое значение. Почему?

Поскольку `tally` копируется по значению (с помощью захвата по умолчанию), версия в лямбда-выражении, по сути, является совершенно другой переменной, которая просто имеет одно и то же имя. Модификации лямбда-копии `tally` не влияют на автоматическую переменную `tally` в `main`. Счет в `main()` инициализируется нулем и никогда не изменяется.

Также есть возможность смешать захват по умолчанию с именованным захватом. Можно, например, выполнить захват по умолчанию по ссылке и скопировать `to_count` по значению, используя следующую формулировку:

```
auto s_counter = [&❶,to_count❷](const char* str) {
  --пропуск--
};
```

Это определяет захват по умолчанию по ссылке ❶ и захват `to_count` по значению ❷.

Хотя выполнение захвата по умолчанию может показаться простым способом, воздержитесь от его использования. Намного лучше объявить захваты явно. Если вы поймали себя на том, что говорите: «Я просто использую захват по умолчанию, потому что в списке слишком много переменных», вам, вероятно, придется реорганизовать код.

Выражения инициализатора в списках захвата

Иногда нужно инициализировать новую переменную в списке захвата. Возможно, переименование захваченной переменной сделало бы более понятным намерение лямбда-выражения. Или, возможно, нужно переместить объект в лямбда-выражение и, следовательно, инициализировать переменную.

Чтобы использовать выражение инициализатора, просто объявите имя новой переменной, за которым следуют знак равенства и значение, которым нужно инициализировать переменную, как показано в листинге 9.23.

Листинг 9.23. Использование выражения инициализатора в лямбда-захвате

```
auto s_counter = [&tally❶,my_char=to_count❷](const char* str) {
  size_t index{}, result{};
  while (str[index]) {
    if (str[index] == my_char❸) result++;
  --пропуск--
};
```

Список захвата содержит простой именованный захват, который хранит `tally` по ссылке ❶. Лямбда-выражение также захватывает `to_count` по значению, но вы решили использовать имя переменной `my_char` вместо этого ❷. Конечно, нужно будет использовать имя `my_char` вместо `to_count` внутри лямбда-выражения ❸.

ПРИМЕЧАНИЕ

Выражение инициализатора в списке захвата также называется захватом инициализации.

Захват this

Иногда лямбда-выражения имеют закрывающий класс. Можно захватить вмещающий объект (на который указывает `this`) по значению или по ссылке, используя либо `[*this]`, либо `[this]` соответственно.

В листинге 9.24 реализован `LambdaFactory`, который генерирует подсчет лямбда-выражений и отслеживает `tally`.

Конструктор `LambdaFactory` принимает один символ и инициализирует им поле `to_count`. Метод `make_lambda` ❶ показывает, как можно захватить `this` по

ссылке ❷ и использовать переменные-члены `to_count` ❸ и `tally` ❹ в лямбда-выражении.

Листинг 9.24. LambdaFactory с использованием захвата this

```
#include <cstdio>
#include <cstdint>

struct LambdaFactory {
  LambdaFactory(char in) : to_count{ in }, tally{} { }
  auto make_lambda() { ❶
    return [this❷](const char* str) {
      size_t index{}, result{};
      while (str[index]) {
        if (str[index] == to_count❸) result++;
        index++;
      }
      tally❹ += result;
      return result;
    };
  }
  const char to_count;
  size_t tally;
};

int main() {
  LambdaFactory factory{ 's' }; ❺
  auto lambda = factory.make_lambda(); ❻
  printf("Tally: %zu\n", factory.tally);
  printf("Sally: %zu\n", lambda("Sally sells seashells by the seashore."));
  printf("Tally: %zu\n", factory.tally);
  printf("Sailor: %zu\n", lambda("Sailor went to sea to see what he could
see."));
  printf("Tally: %zu\n", factory.tally);
}
--------------------------------------------------------------------------------
Tally: 0
Sally: 7
Tally: 7
Sailor: 3
Tally: 10
```

Внутри `main` инициализируется `factory` ❺ и создается лямбда-выражение с помощью метода `make_lambda` ❻. Вывод идентичен листингу 9.19, потому что `this` захватывается по ссылке и состояние `tally` сохраняется при вызовах `lambda`.

Уточняющие примеры

Существует множество возможностей со списками захвата, но если у вас есть команда с основами — захват по значению и по ссылке, сюрпризов не должно быть много. В табл. 9.2 приведены краткие поясняющие примеры, которые можно использовать для справки.

Таблица 9.2. Уточняющие примеры списков лямбда-захвата

Список захвата	Значение
`[&]`	Захват по умолчанию по ссылке
`[&, i]`	Захват по умолчанию по ссылке; захват `i` по значению
`[=]`	Захват по умолчанию по значению
`[=, &i]`	Захват по умолчанию по значению; захват `i` по ссылке
`[i]`	Захват `i` по значению
`[&i]`	Захват `i` по ссылке
`[i, &j]`	Захват `i` по значению, захват `j` по ссылке
`[i=j, &k]`	Захват `j` по значению как `i`, захват `k` по ссылке
`[this]`	Захват вмещающего объекта по ссылке
`[*this]`	Захват вмещающего объекта по значению
`[=, *this, i, &j]`	Захват по умолчанию по значению; захват `this` и `i` по значению; захват `j` по ссылке

Лямбда-выражения с constexpr

Все лямбда-выражения являются `constexpr` до тех пор, пока лямбда-выражение может быть вызвано во время компиляции. При желании можно сделать объявление `constexpr` явным, как показано ниже:

```
[] (int x) constexpr { return x * x; }
```

Нужно отметить лямбда-выражение как `constexpr`, если нужно убедиться, что оно соответствует всем требованиям `constexpr`. На основе C++ 17 это означает отсутствие динамического выделения памяти и вызовов не-`constexpr` функций помимо прочих ограничений. Комитет по стандартизации планирует смягчать эти ограничения с каждым релизом, поэтому, если вы пишете много кода, используя `constexpr`, обязательно ознакомьтесь с последними ограничениями `constexpr`.

std::function

Иногда просто нужно получить единый контейнер для хранения вызываемых объектов. Шаблон класса `std::function` из заголовка `<functions>` представляет собой полиморфную оболочку вокруг вызываемого объекта. Другими словами, это указатель на общую функцию. Можно сохранить статическую функцию, объект функции или лямбда-выражение `в std::function`.

ПРИМЕЧАНИЕ

Класс `function` находится в stdlib. Я расскажу о нем немного раньше времени, потому что он важен для дальнейшего понимания.

С помощью `function` можно:

- вызывать функцию так, чтобы вызывающий объект не знал о реализации функции;
- присваивать, перемещать и копировать;
- иметь пустое состояние, похожее на `nullptr`.

Объявление функции

Чтобы объявить `function`, нужно предоставить один параметр шаблона, содержащий прототип функции вызываемого объекта:

```
std::function<возвращаемый-тип(тип-аргумента-1, тип-аргумента-2, др.)>
```

Шаблон класса `std::function` имеет несколько конструкторов. Конструктор по умолчанию создает `std::function` в пустом режиме, то есть он не содержит вызываемого объекта.

Пустые функции

При вызове `std::function` без содержимого объекта `std::function` выдаст исключение `std::bad_function_call`. Рассмотрим листинг 9.25.

Листинг 9.25. Конструктор std::function по умолчанию и исключение std::bad_function_call

```
#include <cstdio>
#include <functional>

int main() {
    std::function<void()> func; ❶
    try {
        func(); ❷
    } catch(const std::bad_function_call& e) {
        printf("Exception: %s", e.what()); ❸
    }
}
-------------------------------------------------------------------------------
Exception: bad function call ❸
```

`std::function` создается по умолчанию ❶. Параметр шаблона `void()` обозначает функцию, не имеющую аргументов и возвращающую `void`. Поскольку `func` не заполнен вызываемым объектом, он находится в пустом состоянии. При вызове `func` ❷ он генерирует `std::bad_function_call`, который перехватывается и выводится ❸.

Присвоение вызываемого объекта функции

Чтобы присвоить вызываемый объект для функции, можно использовать конструктор или оператор присваивания `function`, как показано в листинге 9.26.

Листинг 9.26. Использование конструктора и оператора присваивания функции

```
#include <cstdio>
#include <functional>

void static_func() { ❶
  printf("A static function.\n");
}

int main() {
  std::function<void()> func { [] { printf("A lambda.\n"); } }; ❷
  func(); ❸
  func = static_func; ❹
  func(); ❺
}
--------------------------------------------------------------------------------
A lambda. ❸
A static function. ❺
```

Объявляется статическая функция `static_func`, которая не принимает аргументов и возвращает `void` ❶. В `main` создается функция с именем `func` ❷. Параметр шаблона указывает, что вызываемый объект, содержащийся в `func`, не принимает аргументов и возвращает `void`. `func` инициализируется с помощью лямбда-выражения, которое выводит сообщение `A lambda`. `func` вызывается сразу после ❸, вызывая содержащееся лямбда-выражение и выводя ожидаемое сообщение. Затем назначается `static_func` для `func`, который заменяет лямбда-выражение, назначенное вами при построении ❹. Затем вызывается функция `func`, которая вызывает `static_func`, а не лямбда-выражение, так что в консоль выводится `A static function` ❺.

Расширенный пример

Можно создать `function` с вызываемыми объектами, если этот объект поддерживает семантику функции, подразумеваемую параметром шаблона `function`.

В листинге 9.27 используется массив экземпляров `std::function`, который заполняется статической функцией, считающей пробелы, объектом функции `CountIf` из листинга 9.12 и лямбда-выражением, вычисляющим длину строки.

Листинг 9.27. Использование массива std::function для перебора унифицированной коллекции вызываемых объектов с различными базовыми типами

```
#include <cstdio>
#include <cstdint>
#include <functional>

struct CountIf {
  --пропуск--
```

```
};

size_t count_spaces(const char* str) {
  size_t index{}, result{};
  while (str[index]) {
    if (str[index] == ' ') result++;
    index++;
  }
  return result;
}

std::function❶<size_t(const char*)❷> funcs[]{
  count_spaces, ❸
  CountIf{ 'e' }, ❹
  [](const char* str) { ❺
    size_t index{};
    while (str[index]) index++;
    return index;
  }
};

auto text = "Sailor went to sea to see what he could see.";

int main() {
  size_t index{};
  for(const auto& func : funcs❻) {
    printf("func #%zu: %zu\n", index++, func(text)❼);
  }
}
--------------------------------------------------------------------------------
func #0: 9 ❸
func #1: 7 ❹
func #2: 44 ❺
```

Массив `std::function` ❶ объявляется со статической длительностью хранения, называемой `funcs`. Аргумент шаблона является прототипом функции для функции, принимающей `constchar*` и возвращающей `size_t` ❷. В массиве `funcs` передается статический указатель на функцию ❸, объект функции ❹ и лямбда-выражение ❺. В `main` используется цикл `for`, основанный на диапазоне, для перебора каждой функции в `funcs` ❻. Каждая функция `func` вызывается с текстом `Sailor went to sea to see what he could see.`, и выводится результат.

Обратите внимание, что с точки зрения `main` все элементы в функциях одинаковы: они просто вызываются строкой с нулевым символом в конце и возвращают `size_t` ❼.

ПРИМЕЧАНИЕ

Использование `function` может привести к издержкам во время выполнения. По техническим причинам `function` может потребовать динамического выделения для хранения вызываемого объекта. Компилятору также трудно оптимизировать вызовы `function`, поэтому часто будет происходить косвенный вызов функций. Косвенные вызовы функций требуют дополнительных указателей разыменования.

Функция main и командная строка

Все программы на C++ должны содержать глобальную функцию с именем `main`. Эта функция определяется как точка входа в программу, которая вызывается при запуске программы. Программы могут принимать любое количество предоставленных средой аргументов, называемых *параметрами командной строки*, при запуске.

Пользователи передают параметры командной строки программам, чтобы настроить их поведение. Вы, вероятно, использовали эту функцию при выполнении программ командной строки, как в команде `copy` (в Linux: `cp`):

```
$ copy file_a.txt file_b.txt
```

При вызове этой команды вы указываете программе скопировать файл `file_a.txt` в файл `file_b.txt`, передав эти значения в качестве параметров командной строки. Как и в случае с программами командной строки, к которым можно привыкнуть, в программы на C++ можно передавать значения в качестве параметров командной строки.

Можно выбрать, будет ли программа обрабатывать параметры командной строки путем различных способов объявления `main`.

Три перегрузки main

Можно получить доступ к параметрам командной строки в `main`, добавив аргументы в объявление `main`.

Существуют три допустимых варианта перегрузки `main`, как показано в листинге 9.28.

Листинг 9.28. Допустимые перегрузки для main

```
int main(); ❶
int main(int argc, char* argv[]); ❷
int main(int argc, char* argv[], параметры-реализации); ❸
```

Первая перегрузка ❶ не требует параметров, как в предыдущих примерах `main()` в этой книге. Используйте этот вариант, если нужно игнорировать любые аргументы, предоставленные программе.

Вторая перегрузка ❷ принимает два параметра, `argc` и `argv`. Первый аргумент, `argc`, является неотрицательным числом, соответствующим количеству элементов в `argv`. Среда рассчитывает это автоматически: не нужно указывать количество элементов в `argc`. Второй аргумент, `argv`, представляет собой аргумент, переданный из среды выполнения.

Третья перегрузка ❸ является расширением второй перегрузки ❷; она принимает произвольное количество дополнительных параметров реализации. Таким образом, целевая платформа может предложить некоторые дополнительные аргументы. Параметры реализации не распространены в современных средах для ПК.

Обычно операционная система передает полный путь к исполняемому файлу. Это поведение зависит от операционной среды. В macOS, Linux и Windows путь к ис-

полняемому файлу является первым аргументом. Формат этого пути зависит от операционной системы. (В главе 17 подробно описаны файловые системы.)

Изучение параметров программ

Давайте создадим программу, чтобы изучить, как операционная система передает параметры в программу. В листинге 9.29 выводится количество аргументов командной строки, а затем выводятся индекс и значение аргументов в каждой строке.

Листинг 9.29. Программа, которая выводит аргументы командной строки. Скомпилируйте эту программу как list_929

```
#include <cstdio>
#include <cstdint>

int main(int argc, char** argv) { ❶
  printf("Arguments: %d\n", argc); ❷
  for(size_t i{}; i<argc; i++) {
    printf("%zu: %s\n", i, argv[i]); ❸
  }
}
```

`main` объявляется с перегрузкой `argc/argv`, которая делает параметры командной строки доступными для программы ❶. Сначала выводится количество аргументов командной строки через `argc` ❷. Затем код циклически просматривает каждый аргумент, выводя его индекс и значение ❸.

Давайте посмотрим на некоторые примеры вывода (на Windows 10 x64). Вот один вызов программы:

```
$ list_929 ❶
Arguments: 1 ❷
0: list_929.exe ❸
```

Здесь не предоставляется никаких дополнительных аргументов командной строки, кроме имени программы, `list_929` ❶. (В зависимости от того, как скомпилирован листинг, нужно заменить его на имя исполняемого файла.) На машине Windows 10 x64 программа получает единственный аргумент ❷, имя исполняемого файла ❸.

И вот еще один вызов:

```
$ list_929 Violence is the last refuge of the incompetent. ❶
Arguments: 9
0: list_929.exe
1: Violence
2: is
3: the
4: last
5: refuge
6: of
7: the
8: incompetent.
```

Здесь приводятся дополнительные аргументы программы: `Violence is the last refuge of the incompetent.` ❶. Из результатов видно, что Windows разделяет командную строку пробелами, в результате чего получается всего девять аргументов.

В основных настольных операционных системах можно заставить операционную систему обрабатывать такую фразу как один аргумент, заключая ее в кавычки, как показано ниже:

```
$ list_929 "Violence is the last refuge of the incompetent."
Arguments: 2
0: list_929.exe
1: Violence is the last refuge of the incompetent.
```

Еще больше примеров

Теперь, когда вы знаете, как обрабатывать ввод из командной строки, давайте рассмотрим более сложный пример. *Гистограмма* — это иллюстрация, которая показывает относительную частоту распределения. Давайте создадим программу, которая вычисляет гистограмму распределения букв аргументов командной строки.

Начните с двух вспомогательных функций, которые определяют, является ли данный символ заглавной или строчной буквой:

```
constexpr char pos_A{ 65 }, pos_Z{ 90 }, pos_a{ 97 }, pos_z{ 122 };
constexpr bool within_AZ(char x) { return pos_A <= x && pos_Z >= x; } ❶
constexpr bool within_az(char x) { return pos_a <= x && pos_z >= x; } ❷
```

Константы `pos_A`, `pos_Z`, `pos_a` и `pos_z` содержат значения ASCII букв A, Z, a и z соответственно (см. таблицу ASCII в табл. 2.4).

Функция `within_AZ` определяет, является ли какой-либо символ `x` заглавной буквой, определяя, находится ли его значение между `pos_A` и `pos_Z` включительно ❶. Функция `within_az` делает то же самое для строчных букв ❷.

Теперь, когда у вас есть некоторые элементы для обработки данных ASCII из командной строки, давайте создадим класс `AlphaHistogram`, который может принимать элементы командной строки и сохранять частоты символов, как показано в листинге 9.30.

Листинг 9.30. AlphaHistogram, который принимает элементы командной строки

```
struct AlphaHistogram {
  void ingest(const char* x); ❶
  void print() const; ❷
private:
  size_t counts[26]{}; ❸
};
```

`AlphaHistogram` будет хранить частоту каждой буквы в массиве `counts` ❸. Этот массив инициализируется нулем всякий раз, когда конструируется `AlphaHistogram`. Метод

`ingest` будет принимать строку с нулевым символом в конце и соответствующим образом обновлять `counts` ❶. Затем метод `print` отобразит информацию о гистограмме, сохраненную в `counts` ❷.

Сначала рассмотрим реализацию `ingest` в листинге 9.31.

Листинг 9.31. Реализация метода ingest

```
void AlphaHistogram::ingest(const char* x) {
  size_t index{}; ❶
  while(const auto c = x[index]) { ❷
    if (within_AZ(c)) counts[c - pos_A]++; ❸
    else if (within_az(c)) counts[c - pos_a]++; ❹
    index++; ❺
  }
}
```

Поскольку `x` является строкой с нулевым символом в конце, ее длина неизвестна раньше времени. Итак, инициализируется переменная `index` ❶ и используется цикл `while` для извлечения одного `char c` за раз ❷. Этот цикл завершится, если `c` равно нулю, что является концом строки. Внутри цикла используется вспомогательная функция `inside_AZ`, чтобы определить, является ли `c` заглавной буквой ❸. Если это так, `pos_A` вычитается из `c`. Это нормализует заглавную букву в интервале от 0 до 25 в соответствии с `counts`. Та же проверка выполняется для строчных букв, используя вспомогательную функцию `inside_az` ❹, и `counts` обновляется в случае, если `c` строчная. Если `c` не является ни строчной, ни прописной, она не влияет на `counts`. Наконец, `index` увеличивается на 1, прежде чем продолжить цикл ❺.

Теперь рассмотрим, как выводить `counts`, это показано в листинге 9.32.

Листинг 9.32. Реализация метода print

```
void AlphaHistogram::print() const {
  for(auto index{ pos_A }; index <= pos_Z; index++) { ❶
    printf("%c: ", index); ❷
    auto n_asterisks = counts[index - pos_A]; ❸
    while (n_asterisks--) printf("*"); ❹
    printf("\n"); ❺
  }
}
```

Чтобы вывести гистограмму, перебирается каждая буква от `A` до `Z` ❶. Внутри цикла сначала печатается буква `index` ❷, а затем определяется, сколько звездочек печатать, путем извлечения правильной буквы из `counts` ❸. Правильное количество звездочек выводится при помощи цикла `while` ❹, а затем выводится завершающий символ новой строки ❺.

Листинг 9.33 показывает `AlphaHistogram` в действии.

Каждый аргумент командной строки после имени программы ❶ перебирается и передается в метод `ingest` объекта `AlphaHistogram` ❷. После поглощения всех аргументов выводится `histogram` ❸. Каждая строка соответствует букве, а звездочки

показывают абсолютную частоту соответствующей буквы. Как видите, фраза `The quick brown fox jumps over the lazy dog` содержит все буквы английского алфавита.

Листинг 9.33. Программа с AlphaHistogram

```
#include <cstdio>
#include <cstdint>

constexpr char pos_A{ 65 }, pos_Z{ 90 }, pos_a{ 97 }, pos_z{ 122 };
constexpr bool within_AZ(char x) { return pos_A <= x && pos_Z >= x; }
constexpr bool within_az(char x) { return pos_a <= x && pos_z >= x; }

struct AlphaHistogram {
  --пропуск--
};

int main(int argc, char** argv) {
  AlphaHistogram hist;
  for(size_t i{ 1 }; i<argc; i++) { ❶
    hist.ingest(argv[i]); ❷
  }
  hist.print(); ❸
}
--------------------------------------------------------------------------------
$ list_933 The quick brown fox jumps over the lazy dog
A: *
B: *
C: *
D: *
E: ***
F: *
G: *
H: **
I: *
J: *
K: *
L: *
M: *
N: *
O: ****
P: *
Q: *
R: **
S: *
T: **
U: **
V: *
W: *
X: *
Y: *
Z: *
```

Статус выхода

Функция `main` может возвращать `int`, соответствующий статусу выхода из программы. То, что представляет значения, определяется средой. Например, в современных настольных системах нулевое возвращаемое значение соответствует успешному выполнению программы. Если оператор возврата явно не задан, компилятор добавляет неявный `return 0`.

Итоги

В этой главе более подробно рассматриваются функции, в том числе то, как объявлять и определять их, как использовать множество доступных ключевых слов для изменения поведения функции, как указывать типы возвращаемых данных, как работает разрешение перегрузки и как принимать переменное число аргументов. После обсуждения того, как принимаются указатели на функции, вы изучили лямбда-выражения и их связь с функциональными объектами. Затем вы узнали о точке входа для программ, функции `main` и о том, как принимать аргументы командной строки.

Упражнения

9.1. Реализуйте шаблон функции свертки со следующим прототипом:

```
template <typename Fn, typename In, typename Out>
constexpr Out fold(Fn function, In* input, size_t length, Out initial);
```

Например, ваша реализация должна поддерживать следующее использование:

```
int main() {
  int data[]{ 100, 200, 300, 400, 500 };
  size_t data_len = 5;
  auto sum = fold([](auto x, auto y) { return x + y; }, data, data_len,
0);
  printf("Sum: %d\n", sum);
}
```

Значение `sum` должно быть 1500. Используйте `fold` для вычисления следующих величин: максимум, минимум и количество элементов со значением больше 200.

9.2. Реализуйте программу, которая принимает произвольное количество аргументов командной строки, считает длину в символах каждого аргумента и выводит гистограмму распределения длин аргументов.

9.3. Реализуйте функцию `all` со следующим прототипом:

```
template <typename Fn, typename In>
constexpr bool all(Fn function, In* input, size_t length);
```

Тип функции `Fn` — это предикат, который поддерживает `bool operator()(In)`. Вся функция должна проверить, возвращает ли функция значение `true` для каждого элемента ввода. Если это так, верните `true`. В противном случае верните `false`.

Например, ваша реализация должна поддерживать следующее использование:

```
int main() {
  int data[]{ 100, 200, 300, 400, 500 };
  size_t data_len = 5;
  auto all_gt100 = all([](auto x) { return x > 100; }, data, data_len);
  if(all_gt100) printf("All elements greater than 100.\n");
}
```

Что еще почитать?

- «Функциональное программирование на C++», Иван Чукич (ДМК-Пресс, 2020)
- «Чистый код. Создание, анализ и рефакторинг», Роберт С. Мартин (Питер, 2019)

ЧАСТЬ 2
Библиотеки и фреймворки C++

Нео: Почему у меня болят глаза?
Морфеус: Ты раньше никогда ими не пользовался.

«Матрица»

Часть 2 знакомит вас с миром библиотек и сред C++, в том числе стандартной библиотекой C++ (stdlib) и библиотеками Boost (Boost Libraries). Последние являются волонтерским проектом с открытым исходным кодом для создания столь необходимых библиотек C++.

В главе 10 вы познакомитесь с несколькими фреймворками тестирования и имитации (мокирования). В отличие от первой части, большинство листингов во второй части являются юнит-тестами. Это дает возможность практиковаться в тестировании кода, а юнит-тесты часто более краткие и выразительные, чем примеры программ на базе `printf`.

В главе 11 подробно рассматриваются умные указатели, которые управляют динамическими объектами и способствуют созданию наиболее мощной модели управления ресурсами на любом языке программирования.

В главе 12 изучается множество утилит, которые реализуют общие задачи программирования.

Глава 13 углубляется в огромный набор контейнеров, которые могут содержать объекты и манипулировать ими.

В главе 14 объясняются итераторы и общий интерфейс, который предоставляют все контейнеры.

В главе 15 рассматриваются строки и строковые операции, которые хранят и манипулируют данными, записанными на естественном языке.

В главе 16 обсуждаются потоки — современный способ выполнения операций ввода и вывода.

Глава 17 раскрывает библиотеку файловой системы, которая предоставляет средства для взаимодействия с файловыми системами.

Глава 18 изучает головокружительный набор алгоритмов, которые запрашивают итераторы и управляют ими.

Глава 19 описывает основные подходы к конкурентности, которая позволяет программам запускать потоки одновременного исполнения.

В главе 20 рассматривается Boost ASIO, кроссплатформенная библиотека для сетевого и низкоуровневого программирования ввода/вывода с использованием асинхронного подхода.

Глава 21 представляет несколько прикладных структур, которые реализуют стандартные структуры, необходимые в повседневном прикладном программировании.

Часть 2 станет хорошим кратким справочником, но первый раз прочтите ее главы по порядку.

10 Тестирование

«Компьютер... мог получить этот портрет, только если его заказали недавно, у гражданского компьютера, никак не связанного с Международным флотом ... Кто в нашей школе подписал подобное разрешение?»
«Если честно, сэр, я не знаю. А игровая программа не может нам рассказать, она так устроена. Да, скорее всего, она и сама не знает».

Орсон Скотт Кард, «Игра Эндера»

Для тестирования ПО существуют множество способов. Связующей нитью всех этих методов тестирования, является то, что каждый тест обеспечивает некоторый ввод данных в код и результаты теста оцениваются на наличие соответствия. Характер среды, объем исследования и форма оценки сильно различаются в зависимости от типа тестирования. В этой главе рассказывается, как выполнить тестирование с использованием нескольких различных сред, однако этот материал можно использовать для других вариантов тестирования. Перед тем как погрузиться, давайте кратко рассмотрим несколько видов тестирования.

Юнит-тестирование

Юнит-тесты подтверждают, что целенаправленный, сплоченный набор кода — *модуль* (unit), такой как функция или класс, — ведет себя именно так, как задумал программист. Хорошие юнит-тесты изолируют тестируемый модуль от его зависимостей. Иногда это трудно сделать: модуль может зависеть от других модулей. В таких ситуациях используются имитации (моки, mocks), чтобы заменить эти зависимости. Имитация — это фальшивые объекты, которые используются исключительно во время тестирования, чтобы предоставить детальный контроль над поведением зависимостей

модуля во время теста. Имитация также может записывать, как модуль взаимодействовал с ней, так что можно проверить, взаимодействует ли модуль с его зависимостями, как и ожидалось. Также можно использовать имитации для повторения редких событий, таких как нехватка памяти, запрограммировав их на выдачу исключения.

Интеграционное тестирование

Тестирование совокупности модулей вместе называется *интеграционным тестированием*. Интеграционные тесты могут также относиться к тестированию взаимодействия между программным и аппаратным обеспечением, с которым часто сталкиваются системные программисты. Интеграционные тесты — это важный слой поверх юнит-тестов, поскольку они гарантируют, что написанное программное обеспечение работает вместе как система. Эти тесты дополняют, но не заменяют юнит-тесты.

Приемочное тестирование

Приемочные тесты гарантируют, что ПО соответствует всем требованиям клиентов. Высокопроизводительные команды разработчиков ПО могут использовать приемочные тесты для руководства разработкой. После того как все приемочные испытания пройдут, ПО будет внедрено. Поскольку эти приемочные тесты становятся частью кодовой базы, существует встроенная защита от рефакторинга или регрессии функций — нарушения существующей функции в процессе добавления новой.

Тестирование производительности

Тесты производительности оценивают, соответствует ли ПО требованиям эффективности, таким как скорость выполнения или потребление памяти/энергии. Оптимизация кода является принципиально эмпирическим упражнением. Вы можете (и должны) иметь представление о том, какие части кода вызывают узкие места в производительности, но не можете знать наверняка, если не измерите это. Кроме того, нельзя знать, улучшают ли код изменения, которые вы реализуете с целью оптимизации, пока это не будет измерено снова. Можно использовать тесты производительности в качестве инструментов кода и обеспечивать соответствующие меры. Инструментарий — это методика измерения производительности продукта, обнаружения ошибок и регистрации выполнения программы. Иногда клиенты предъявляют строгие требования к производительности (например, вычисления не могут занимать более 100 миллисекунд или система не может выделить более 1 МБ памяти). Можно автоматизировать тестирование таких требований и убедиться, что будущие изменения кода не нарушат их.

Тестирование кода может быть абстрактным сухим предметом. Чтобы избежать этого, в следующем разделе представлен расширенный пример, который предоставляет контекст для обсуждения.

Расширенный пример: AutoBrake

Предположим, вы пишете софт для автономного транспортного средства. ПО команды очень сложное и включает в себя сотни тысяч строк кода. Все программное решение состоит из нескольких двоичных файлов. Чтобы развернуть ПО, нужно загрузить двоичные файлы в машину (используя относительно длительный процесс). Внесение изменений в код, компиляция, загрузка и выполнение его в реальном времени занимают несколько часов.

Монументальная задача написания всего софта автомобиля разбита на команды. Каждая команда отвечает за *обслуживание* — руление, аудио/видео или обнаружение расположения автомобиля. Сервисы взаимодействуют друг с другом через служебную шину, где каждый сервис публикует события. Другие службы подписываются на эти события по мере необходимости. Этот шаблон проектирования называется *архитектурой служебной шины*.

Ваша команда отвечает за службу автономного торможения. Служба должна определить, должно ли произойти столкновение, и если это так, сообщить машине о торможении. Ваш сервис подписывается на два типа событий: из класса `SpeedUpdate`, который сообщает, что скорость автомобиля изменилась, и класса `CarDetected`, который сообщает, что перед вами была обнаружена какая-то другая машина. Ваша система отвечает за публикацию `BrakeCommand` на служебной шине при обнаружении неизбежного столкновения. Эти классы приведены в листинге 10.1.

Листинг 10.1. Классы POD, с которыми взаимодействует ваша служба

```
struct SpeedUpdate {
  double velocity_mps;
};

struct CarDetected {
  double distance_m;
  double velocity_mps;
};

struct BrakeCommand {
  double time_to_collision_s;
};
```

`BrakeCommand` публикуется с помощью объекта `ServiceBus`, у которого есть метод `publish`:

```
struct ServiceBus {
  void publish(const BrakeCommand&);
  --пропуск--
};
```

Ведущий архитектор хочет, чтобы вы представили метод `observe`, чтобы можно было подписаться на события `SpeedUpdate` и `CarDetected` на служебной шине. Вы решаете создать класс `AutoBrake`, который инициализируется в точке входа в программу. Класс `AutoBrake` будет хранить ссылку на метод `publish` служебной шины

и подписываться на события `SpeedUpdate` и `CarDetected` через свой метод `observe`, как показано в листинге 10.2.

Листинг 10.2. Класс AutoBrake, предоставляющий службу автоматического торможения

```
template <typename T>
struct AutoBrake {
  AutoBrake(const T& publish);
  void observe(const SpeedUpdate&);
  void observe(const CarDetected&);
private:
  const T& publish;
  --пропуск--
};
```

Рисунок 10.1 обобщает взаимосвязь между сервисной шиной `ServiceBus`, системой автоматического торможения `AutoBrake` и другими сервисами.

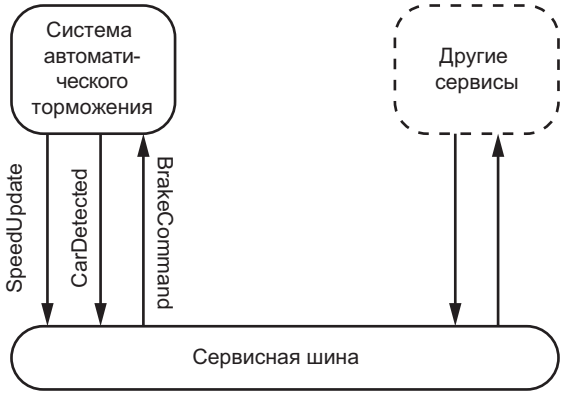


Рис. 10.1. Высокоуровневое описание взаимодействия между сервисами и сервисной шиной

Сервис интегрируется в ПО автомобиля, получая что-то вроде кода в листинге 10.3.

Листинг 10.3. Пример точки входа с использованием сервиса AutoBrake

```
--пропуск--
int main() {
  ServiceBus bus;
  AutoBrake auto_brake{ [&bus❶] (const auto& cmd) {
                          bus.publish(cmd); ❷
                        }
};
while (true) {    // Цикл событий сервисной шины
  auto_brake.observe(SpeedUpdate{ 10L }); ❸
  auto_brake.observe(CarDetected{ 250L, 25L }); ❹
  }
}
```

`AutoBrake` создается с лямбда-выражением, которое захватывает ссылку на `ServiceBus` ❶. Все детали того, как `AutoBrake` решает, когда тормозить, полностью скрыты от других команд. Сервисная шина опосредует все межсервисные коммуникации. Любые команды просто передаются от `AutoBrake` непосредственно к `ServiceBus` ❷. В цикле событий `ServiceBus` может передавать объекты `SpeedUpdate` ❸ и `CarDetected` ❹ методу `observe` в `auto_brake`.

Реализация AutoBrake

Концептуально простой способ реализации `AutoBrake` состоит в том, чтобы выполнять итерацию между написанием некоторого кода, компиляцией производственного двоичного файла, загрузкой его в машину и тестированием функциональности вручную. Такой подход может привести к программным (и машинным) сбоям и вынудит потратить много времени. Лучшим подходом являются написание кода, компиляция двоичного кода юнит-теста и запуск его в среде разработки ПК. Эти шаги можно итерировать быстрее; если вы достаточно уверены в том, что написанный код работает, как задумано, можно провести ручной тест на настоящем автомобиле.

Двоичный тестовый модуль будет простым консольным приложением, предназначенным для операционной системы ПК. В двоичном юнит-тесте запускается набор юнит-тестов, которые передают определенные входные данные в `AutoBrake` и утверждают, что он дает ожидаемые результаты.

После консультации с управленческой командой вы собрали следующие требования:

- `AutoBrake` будет считать начальную скорость автомобиля нулевой;
- `AutoBrake` должен иметь настраиваемый порог чувствительности, основанный на прогнозе количества секунд до столкновения. Чувствительность должна быть не менее 1 секунды. Чувствительность по умолчанию составляет 5 секунд;
- `AutoBrake` должен сохранять скорость автомобиля между наблюдениями `SpeedUpdate`;
- каждый раз, когда `AutoBrake` наблюдает событие `CarDetected`, он должен опубликовать `BrakeCommand`, если столкновение прогнозируется за меньшее время, чем настроенный порог чувствительности.

Поскольку вы собрали такой нетронутый список требований, следующий шаг — попытаться внедрить службу автоматического торможения с использованием *разработки через тестирование* (test-driven development, TDD).

ПРИМЕЧАНИЕ

Поскольку эта книга посвящена C++, а не физике, `AutoBrake` работает только тогда, когда автомобиль находится прямо перед вами.

Разработка через тестирование

В какой-то момент в истории внедрения юнит-тестирования некоторые бесстрашные инженеры-программисты подумали: «Если я знаю, что собираюсь написать кучу юнит-тестов для этого класса, почему бы сначала не написать тесты?» Этот способ написания программного обеспечения, известный как TDD, лежит в основе одной из великих религиозных войн в сообществе разработчиков программного обеспечения. Vim или Emacs? Табуляция или пробелы? Использовать TDD или не использовать TDD? Эта книга смиренно воздерживается от оценки этих вопросов. Но мы будем использовать TDD, потому что она естественно вписывается в обсуждение юнит-тестирования.

Преимущества TDD

Процесс написания теста, который кодирует требование *перед* реализацией решения, является фундаментальной идеей TDD. Сторонники говорят, что код, написанный таким образом, имеет тенденцию быть более модульным, надежным, чистым и хорошо спроектированным. Написание хороших тестов — лучший способ документировать код для других разработчиков. Хороший набор тестов — это полностью рабочий набор примеров, который никогда не выйдет из синхронизации. Он защищает от регрессии в функциональности при добавлении новых функций.

Юнит-тесты также служат отличным способом отправки отчетов об ошибках при написании неудачного тестового модуля. Как только ошибка исправлена, она останется исправленной, потому что юнит-тест и код, который исправляет ошибку, становятся частью набора тестов.

Красный-зеленый-рефакторинг

У практикующих TDD имеется мантра: *красный, зеленый, рефакторинг*. Красный — это первый шаг, он означает, что нужно выполнить неуспешный тест. Это делается по нескольким причинам, главная из которых — убедиться, что вы что-то действительно тестируете. Вас может удивить, насколько часто случайно разрабатывается тест, который не делает никаких утверждений. Затем реализуется код, который проходит тест. Ни больше ни меньше. Это превращает тест из красного в зеленый. Теперь, когда у вас есть рабочий код и пройденный тест, можно реорганизовать рабочий код. Рефакторинг означает реструктуризацию существующего кода без изменения его функциональности. Например, можно найти более элегантный способ написания того же кода, заменить код сторонней библиотекой или переписать код, чтобы улучшить характеристики производительности.

Если вы случайно что-то сломаете, то сразу об этом узнаете, потому что набор тестов скажет об этом. Затем вы продолжите реализовывать оставшуюся часть класса, используя TDD. Можно работать на пороге столкновения и дальше.

Написание скелетного класса AutoBrake

Прежде чем писать тесты, нужно написать *скелетный класс*, который реализует интерфейс, но не предоставляет никаких функциональных возможностей. Это полезно в TDD, потому что нельзя скомпилировать тест без оболочки тестируемого класса.

Рассмотрим скелетный класс `AutoBrake` в листинге 10.4.

Листинг 10.4. Скелетный класс AutoBrake

```
struct SpeedUpdate {
  double velocity_mps;
};

struct CarDetected {
  double distance_m;
  double velocity_mps;
};

struct BrakeCommand {
  double time_to_collision_s;
};

template <typename T>
struct AutoBrake {
  AutoBrake(const T& publish❶) : publish{ publish } { }
  void observe(const SpeedUpdate& cd) { } ❷
  void observe(const CarDetected& cd) { } ❸
  void set_collision_threshold_s(double x) { ❹
    collision_threshold_s = x;
  }
  double get_collision_threshold_s() const { ❺
    return collision_threshold_s;
  }
  double get_speed_mps() const { ❻
    return speed_mps;
  }
private:
  double collision_threshold_s;
  double speed_mps;
  const T& publish;
};
```

Класс `AutoBrake` имеет единственный конструктор, который принимает параметр шаблона `publish` ❶, сохраняющегося в члене `const`. Одно из требований гласит, что публикация будет вызываться с помощью `BrakeCommand`. Использование параметра шаблона `T` позволяет программировать общий случай для любого типа, который поддерживает вызов с `BrakeCommand`. Предоставляются две разные функции наблюдения: по одной для каждого типа событий, на которые нужно подписаться ❷❸. Поскольку это просто скелетный класс, в теле нет никаких инструкций. Вам просто нужен класс, который предоставляет соответствующие методы и компилируется без ошибок. Поскольку методы возвращают `void`, даже не требуется оператор `return`.

Пришло время реализовать сеттер ❹ и геттер ❺. Эти методы опосредуют взаимодействие с закрытой переменной-членом `collision_threshold_s`. Одно из требований подразумевает инвариант класса относительно допустимых значений для `collision_threshold_s`. Поскольку это значение может измениться после построения, нельзя просто использовать конструктор для установки инварианта класса. Необходим

способ применения этого инварианта класса в течение всего времени жизни объекта. Можно использовать сеттер для проверки перед тем, как класс установит значение члена. Геттер позволяет прочитать значение `collision_threshold_s` без разрешения изменения. Это усиливает своего рода внешнее постоянство.

Наконец, имеется геттер для `speed_mps` ❻ без соответствующего сеттера. Это похоже на превращение `speed_mps` в публичный член с тем важным отличием, что было бы возможно изменить `speed_mps` из внешнего класса, если бы он был публичным.

Утверждение: строительные блоки юнит-тестов

Вжнейшим компонентом юнит-теста является *утверждение*, которое проверяет, что какое-либо условие выполнено. Если условие не выполнено, тест, содержащий его, не пройден.

В листинге 10.5 реализована функция `assert_that`, которая выдает исключение с сообщением об ошибке, когда какой-либо логический `statement` имеет значение `false`.

Листинг 10.5. Программа с assert_that (выходные данные из двоичного файла, скомпилированного в GCC v7.1.1)

```
#include <stdexcept>
constexpr void assert_that(bool statement, const char* message) {
  if (!statement❶) throw std::runtime_error{ message }; ❷
}

int main() {
  assert_that(1 + 2 > 2, "Something is profoundly wrong with the universe."); ❸
  assert_that(24 == 42, "This assertion will generate an exception."); ❹
}
--------------------------------------------------------------------------------
terminate called after throwing an instance of 'std::runtime_error'
what(): This assertion will generate an exception. ❹
```

Функция `assert_that` проверяет, является ли параметр `statement` ❶ ложным, и в этом случае она выдает исключение с параметром `message` ❷. Первое утверждение проверяет, что $1 + 2 > 2$, что является истиной ❸. Второе утверждение проверяет, что $24 == 42$, что неверно, и потому выдается необработанное исключение ❹.

Требование: начальная скорость равна нулю

Рассмотрим первое требование, чтобы начальная скорость автомобиля была равна нулю. Перед реализацией этой функции в `AutoBrake` необходимо написать юнит-тест, который кодирует это требование. Юнит-тест реализуется как функция, которая создает `AutoBrake`, проверяет класс и делает утверждения о результатах. Листинг 10.6 содержит юнит-тест, который кодирует требование о том, что начальная скорость равна нулю.

Сначала создается `AutoBrake` с пустой функцией `BrakeCommandpublish` ❶. Этот юнит-тест касается только начального значения `AutoBrake` для скорости автомо-

биля. Поскольку этот юнит-тест не связан с тем, как и когда `AutoBrake` публикует `BrakeCommand`, ему передается самый простой аргумент, который все равно будет компилироваться.

Листинг 10.6. Юнит-тест, кодирующий требование, чтобы начальная скорость была нулевой

```
void initial_speed_is_zero() {
  AutoBrake auto_brake{ [](const BrakeCommand&) {} }; ❶
  assert_that(auto_brake.get_speed_mps() == 0L, "speed not equal 0"); ❷
}
```

ПРИМЕЧАНИЕ

Тонкая, но важная особенность юнит-тестов заключается в том, что если не нужна некоторая зависимость тестируемого модуля, можно просто предоставить пустую реализацию, которая выполняет какое-то безобидное поведение по умолчанию. Эту пустую реализацию иногда называют заглушкой (стабом).

В `initial_speed_is_zero` нужно только проверить утверждение, что начальная скорость автомобиля равна нулю ❷. Для этого используется геттер `get_speed_mps` и возвращаемое значение сравнивается с 0. Это все, что нужно сделать; `assert` выдаст исключение, если начальная скорость не равна нулю.

Теперь необходим способ запуска юнит-тестов.

Окружение теста

Окружение теста — это код, который выполняет юнит-тесты. Можно создать окружение, которое будет вызывать функции юнит-теста, такие как `initial_speed_is_zero`, и корректно обрабатывать ошибочные утверждения. Рассмотрим окружение теста `run_test` в листинге 10.7.

Листинг 10.7. Окружение теста

```
#include <exception>
--пропуск--
void run_test(void(*unit_test)(), const char* name) {
  try {
    unit_test(); ❶
    printf("[+] Test %s successful.\n", name); ❷
  } catch (const std::exception& e) {
    printf("[-] Test failure in %s. %s.\n", name, e.what()); ❸
  }
}
```

Окружение `run_test` принимает юнит-тест в виде указателя на функцию с именем `unit_test` и вызывает его в операторе `try-catch` ❶. Пока `unit_test` не генерирует исключение, `run_test` будет выводить дружеское сообщение о том, что юнит-тест пройден до возврата ❷. Если выброшено какое-либо исключение, тест завершается неудачно и выводится соответствующее сообщение ❸.

Чтобы создать *программу юнит-тестирования*, которая будет запускать все юнит-тесты, поместите тестовое окружение `run_test` в основную функцию новой программы. В целом программа юнит-тестирования выглядит как листинг 10.8.

Листинг 10.8. Программа юнит-тестирования

```
#include <stdexcept>

struct SpeedUpdate {
  double velocity_mps;
};

struct CarDetected {
  double distance_m;
  double velocity_mps;
};

struct BrakeCommand {
  double time_to_collision_s;
};

template <typename T>
struct AutoBrake {
  --пропуск--
};

constexpr void assert_that(bool statement, const char* message) {
  if (!statement) throw std::runtime_error{ message };
}

void initial_speed_is_zero() {
  AutoBrake auto_brake{ [](const BrakeCommand&) {} };
  assert_that(auto_brake.get_speed_mps() == 0L, "speed not equal 0");
}

void run_test(void(*unit_test)(), const char* name) {
  try {
    unit_test();
    printf("[+] Test %s successful.\n", name);
  } catch (const std::exception& e) {
    printf("[-] Test failure in %s. %s.\n", name, e.what());
  }
}

int main() {
  run_test(initial_speed_is_zero, "initial speed is 0"); ❶
}
--------------------------------------------------------------------------------
[-] Test failure in initial speed is 0. speed not equal 0. ❶
```

После компиляции и запуска этого двоичного файла юнит-теста вы увидите, что юнит-тест `initial_speed_is_zero` завершается с информативным сообщением ❶.

ПРИМЕЧАНИЕ

Поскольку элемент `AutoBrake speed_mps` не инициализирован в листинге 10.8, эта программа имеет неопределенное поведение. На самом деле нет уверенности, что тест провалится. Решение, конечно, заключается в том, что не стоит писать программы с неопределенным поведением.

Прохождение теста

Чтобы передать `initial_speed_is_zero`, требуется лишь инициализировать `speed_mps` нулем в конструкторе `AutoBrake`:

```
template <typename T>
struct AutoBrake {
  AutoBrake(const T& publish) : speed_mps{}❶, publish{ publish } { }
  --пропуск--
};
```

Просто добавьте в инициализацию нуль ❶. Теперь, если обновить, скомпилировать и запустить программу юнит-тестирования из листинга 10.8, получится более приятный вывод:

```
[+] Test initial speed is 0 successful.
```

Требование: порог столкновения по умолчанию равен пяти

Порог столкновения по умолчанию должен быть равен 5. Рассмотрим юнит-тест в листинге 10.9.

Листинг 10.9. Юнит-тест, кодирующий требование, чтобы начальная скорость была равна нулю

```
void initial_sensitivity_is_five() {
  AutoBrake auto_brake{ [](const BrakeCommand&) {} };
  assert_that(auto_brake.get_collision_threshold_s() == 5L,
              "sensitivity is not 5");
}
```

Этот тест можно вставить в тестовую программу, как показано в листинге 10.10.

Листинг 10.10. Добавление теста initial-sensitivity-is-5 к окружению теста

```
--пропуск--
int main() {
  run_test(initial_speed_is_zero, "initial speed is 0");
  run_test(initial_sensitivity_is_five, "initial sensitivity is 5");
}
-------------------------------------------------------------------------------
[+] Test initial speed is 0 successful.
[-] Test failure in initial sensitivity is 5. sensitivity is not 5.
```

Как и ожидалось, в листинге 10.10 видно, что `initial_speed_is_zero` все еще проходит, а новый тест `initial_sensitivity_is_five` завершается неудачей.

Теперь сделайте так, чтобы тест успешно проходил. Добавьте соответствующий инициализатор члена в `AutoBrake`, как показано в листинге 10.11.

Листинг 10.11. Обновление AutoBrake для удовлетворения порогового значения столкновения

```
template <typename T>
struct AutoBrake {
  AutoBrake(const T& publish)
    : collision_threshold_s{ 5 }, ❶
      speed_mps{},
      publish{ publish } { }
    --пропуск--
};
```

Инициализатор нового члена ❶ устанавливает для `collision_threshold_s` значение 5. Перекомпилируя тестовую программу, вы можете заметить, что `initial_sensitivity_is_five` теперь проходит:

```
[+] Test initial speed is 0 successful.
[+] Test initial sensitivity is 5 successful.
```

Затем обработайте инвариант класса: чувствительность должна быть больше 1.

Требование: чувствительность всегда должна быть больше 1

Чтобы кодировать ошибки проверки чувствительности с использованием исключений, можно создать тест, который ожидает, что при значении `collision_threshold_s` меньше 1 будет выдано исключение, как показано в листинге 10.12.

Листинг 10.12. Тест, кодирующий требование, что чувствительность всегда должна быть больше 1

```
void sensitivity_greater_than_1() {
  AutoBrake auto_brake{ [](const BrakeCommand&) {} };
  try {
    auto_brake.set_collision_threshold_s(0.5L); ❶
  } catch (const std::exception&) {
    return; ❷
  }
  assert_that(false, "no exception thrown"); ❸
}
```

Ожидается, что метод `set_collision_threshold_s` в `auto_brake` выдаст исключение при вызове со значением 0, 5 ❶. Если это так, исключение перехватывается, а тест немедленно завершается ❷. Если `set_collision_threshold_s` не выдает исключение, утверждение проваливается с сообщением `no exception thrown` ❸.

Затем добавьте `sensitivity_greater_than_1` к окружению теста, как показано в листинге 10.13.

Как и ожидалось, новый юнит-тест не пройден ❶.

Листинг 10.13. Добавление set_collision_threshold_s в окружение теста

```
--пропуск--
int main() {
  run_test(initial_speed_is_zero, "initial speed is 0");
  run_test(initial_sensitivity_is_five, "initial sensitivity is 5");
  run_test(sensitivity_greater_than_1, "sensitivity greater than 1"); ❶
}
-------------------------------------------------------------------------------
[+] Test initial speed is 0 successful.
[+] Test initial sensitivity is 5 successful.
[-] Test failure in sensitivity greater than 1. no exception thrown. ❶
```

Можно реализовать проверку, которая позволит пройти тест, как показано в листинге 10.14.

Листинг 10.14. Обновление метода set_collision_threshold в AutoBrake для проверки его входных данных

```
#include <exception>
--пропуск--
template <typename T>
struct AutoBrake {
  --пропуск--
  void set_collision_threshold_s(double x) {
    if (x < 1) throw std:: invalid_argument{ "Collision less than 1." };
    collision_threshold_s = x;
  }
}
```

Перекомпиляция и выполнение пакета юнит-тестов делают тест зеленым:

```
[+] Test initial speed is 0 successful.
[+] Test initial sensitivity is 5 successful.
[+] Test sensitivity greater than 1 successful.
```

Далее нужно убедиться, что `AutoBrake` сохраняет скорость автомобиля между каждым `SpeedUpdate`.

Требование: сохранить скорость автомобиля между обновлениями

Юнит-тест в листинге 10.15 кодирует требование, чтобы `AutoBrake` сохранял скорость автомобиля.

Листинг 10.15. Кодирование требования, что AutoBrake сохраняет скорость автомобиля

```
void speed_is_saved() {
  AutoBrake auto_brake{ [](const BrakeCommand&) {} }; ❶
  auto_brake.observe(SpeedUpdate{ 100L }); ❷
  assert_that(100L == auto_brake.get_speed_mps(), "speed not saved to 100"); ❸
  auto_brake.observe(SpeedUpdate{ 50L });
  assert_that(50L == auto_brake.get_speed_mps(), "speed not saved to 50");
  auto_brake.observe(SpeedUpdate{ 0L });
  assert_that(0L == auto_brake.get_speed_mps(), "speed not saved to 0");
}
```

После создания `AutoBrake` ❶ `SpeedUpdate` передается с `velocity_mps`, равным 100, в его метод `observe` ❷. Затем возвращается скорость из `auto_brake` с помощью метода `get_speed_mps` и ожидается, что она равна 100 ❸.

ПРИМЕЧАНИЕ

Как правило, в каждом тесте должно быть одно утверждение. Этот тест нарушает строжайшее толкование этого правила, но не нарушает его дух. Во всех утверждениях рассматривается одно и то же связное требование, заключающееся в том, что скорость сохраняется при каждом обнаружении `SpeedUpdate`.

Тест из листинга 10.15 добавляется в окружение теста обычным способом, как показано в листинге 10.16.

Листинг 10.16. Добавление юнит-теста для сохранения скорости в окружение теста

```
--пропуск--
int main() {
  run_test(initial_speed_is_zero, "initial speed is 0");
  run_test(initial_sensitivity_is_five, "initial sensitivity is 5");
  run_test(sensitivity_greater_than_1, "sensitivity greater than 1");
  run_test(speed_is_saved, "speed is saved"); ❶
}
--------------------------------------------------------------------------------
[+] Test initial speed is 0 successful.
[+] Test initial sensitivity is 5 successful.
[+] Test sensitivity greater than 1 successful.
[-] Test failure in speed is saved. speed not saved to 100. ❶
```

Неудивительно, что новый тест не прошел ❶. Чтобы пройти этот тест, нужно реализовать соответствующую функцию `observe`:

```
template <typename T>
struct AutoBrake {
  --пропуск--
  void observe(const SpeedUpdate& x) {
    speed_mps = x.velocity_mps; ❶
  }
};
```

`speed_mps` извлекается из `SpeedUpdate` и сохраняется в переменную-член `speed_mps` ❶. Перекомпиляция тестового двоичного файла показывает, что юнит-тест теперь проходит:

```
[+] Test initial speed is 0 successful.
[+] Test initial sensitivity is 5 successful.
[+] Test sensitivity greater than 1 successful.
[+] Test speed is saved successful.
```

Наконец, требуется, чтобы `AutoBrake` мог вычислить правильное время для столкновения и при необходимости опубликовать `BrakeCommand`, используя функцию `publish`.

Требование: AutoBrake публикует команду BrakeCommand при обнаружении столкновения

Соответствующие уравнения для вычисления времени столкновения приходят непосредственно из физики средней школы. Сначала рассчитывается относительная скорость выбранного автомобиля к скорости обнаруженного автомобиля:

$$Velocity_{Relative} = Velocity_{OurCar} - Velocity_{OtherCar}$$

Если относительная скорость постоянна и положительна, автомобили в конечном итоге столкнутся. Можно вычислить время для такого столкновения следующим образом:

$$Time_{Collision} = Distance / Velocity_{Relative}$$

Если `TimeCollision` больше нуля и меньше или равно `collision threshold_s`, вызывается `publish` в `BrakeCommand`. Юнит-тест в листинге 10.17 устанавливает порог столкновения в 10 секунд, а затем мониторит события, которые указывают на сбой.

Листинг 10.17. Юнит-тест для событий торможения

```
void alert_when_imminent() {
  int brake_commands_published{}; ❶
  AutoBrake auto_brake{
    [&brake_commands_published❷](const BrakeCommand&) {
      brake_commands_published++; ❸
  } };
  auto_brake.set_collision_threshold_s(10L); ❹
  auto_brake.observe(SpeedUpdate{ 100L }); ❺
  auto_brake.observe(CarDetected{ 100L, 0L }); ❻
  assert_that(brake_commands_published == 1, "brake commands published not one"); ❼
}
```

Здесь локальная переменная `brake_commands_published` инициализируется нулем ❶. Это позволит отслеживать количество вызовов обратного вызова `publish`. Эта локальная переменная передается по ссылке в лямбда-выражение, используемое для создания `auto_brake` ❷. Обратите внимание, что `brake_commands_published` увеличивается ❸. Поскольку лямбда-выражение использует захват по ссылке, можно проверить значение `brake_commands_published` позже в юнит-тесте. Затем значение `set_collision_threshold` устанавливается равным 10 ❹. Скорость автомобиля обновляется до 100 метров в секунду ❺, а затем обнаруживается, что автомобиль на расстоянии 100 метров движется со скоростью 0 метров в секунду (он остановлен) ❻. Класс `AutoBrake` должен определить, что столкновение произойдет через 1 секунду. Это должно вызвать обратный вызов, который будет увеличивать `brake_commands_published`. Утверждение ❼ гарантирует, что обратный вызов происходит ровно один раз.

После добавления в `main` скомпилируйте и запустите код, чтобы получить новый красный тест:

```
[+] Test initial speed is 0 successful.
[+] Test initial sensitivity is 5 successful.
[+] Test sensitivity greater than 1 successful.
[+] Test speed is saved successful.
[-] Test failure in alert when imminent. brake commands published not one.
```

Можно реализовать код для прохождения этого теста. В листинге 10.18 представлен весь код, необходимый для выдачи команд торможения.

Листинг 10.18. Код, реализующий функцию торможения

```
template <typename T>
struct AutoBrake {
  --пропуск--
  void observe(const CarDetected& cd) {
    const auto relative_velocity_mps = speed_mps - cd.velocity_mps; ❶
    const auto time_to_collision_s = cd.distance_m / relative_velocity_mps; ❷
    if (time_to_collision_s > 0 && ❸
        time_to_collision_s <= collision_threshold_s ❹) {
        publish(BrakeCommand{ time_to_collision_s }); ❺
    }
  }
};
```

Сначала вычисляется относительная скорость ❶. Затем это значение используется для вычисления времени до столкновения ❷. Если это значение положительное ❸ и меньше или равно порогу столкновения ❹, `BrakeCommand` публикуется ❺.

Перекомпиляция и запуск пакета юнит-тестов дает успех:

```
[+] Test initial speed is 0 successful.
[+] Test initial sensitivity is 5 successful.
[+] Test sensitivity greater than 1 successful.
[+] Test speed is saved successful.
[+] Test alert when imminent successful.
```

Наконец, нужно убедиться, что `AutoBrake` не будет вызывать публикацию с помощью `BrakeCommand`, если столкновение произойдет позже, чем `collision_threshold_s`. Можно повторно использовать юнит-тест `alert_when_imminent`, как показано в листинге 10.19.

Листинг 10.19. Автомобиль не выдает BrakeCommand, если столкновение не ожидается в пределах порога столкновения

```
void no_alert_when_not_imminent() {
  int brake_commands_published{};
  AutoBrake auto_brake{
    [&brake_commands_published](const BrakeCommand&) {
      brake_commands_published++;
  } };
  auto_brake.set_collision_threshold_s(2L);
  auto_brake.observe(SpeedUpdate{ 100L });
  auto_brake.observe(CarDetected{ 1000L, 50L });
  assert_that(brake_commands_published == 0 ❶, "brake command published");
}
```

Это меняет настройки. Порог автомобиля установлен на 2 секунды со скоростью 100 метров в секунду. Автомобиль обнаружен на расстоянии 1000 метров со скоростью 50 метров в секунду. Класс `AutoBrake` должен прогнозировать столкновение через 20 секунд, что превышает двухсекундный порог. Утверждение также изменится ❶.

После добавления этого теста в `main` и запуска пакета юнит-тестов вы получите следующее:

```
[+] Test initial speed is 0 successful.
[+] Test initial sensitivity is 5 successful.
[+] Test sensitivity greater than 1 successful.
[+] Test speed is saved successful.
[+] Test alert when imminent successful.
[+] Test no alert when not imminent successful. ❶
```

Для прохождения этого теста уже существует весь необходимый код ❶. Отсутствие неуспешного теста с самого начала обходит мантру «красный, зеленый, рефакторинг», но это нормально. Этот тестовый пример тесно связан с `alert_when_imminent`. Смысл TDD не в догматической приверженности строгим правилам. TDD — это набор довольно свободных руководств, которые помогут написать лучшее программное обеспечение.

Добавление интерфейса Service-Bus

Класс `AutoBrake` имеет несколько зависимостей: `CarDetected`, `SpeedUpdated` и общую зависимость от некоторого объекта `publish`, вызываемого одним параметром `BrakeCommand`. Классы `CarDetected` и `SpeedUpdated` являются типами простых данных, которые легко использовать непосредственно в юнит-тестах. Объект `publish` немного сложнее инициализировать, но благодаря лямбда-выражениям все не так плохо.

Предположим, нужно провести рефакторинг сервисной шины. Требуется принять `std::function` для подписки на каждую службу, как в новом интерфейсе `IServiceBus` в листинге 10.20.

Листинг 10.20. Интерфейс IServiceBus

```
#include <functional>

using SpeedUpdateCallback = std::function<void(const SpeedUpdate&)>;
using CarDetectedCallback = std::function<void(const CarDetected&)>;

struct IServiceBus {
  virtual ~IServiceBus() = default;
  virtual void publish(const BrakeCommand&) = 0;
  virtual void subscribe(SpeedUpdateCallback) = 0;
  virtual void subscribe(CarDetectedCallback) = 0;
};
```

Поскольку `IServiceBus` является интерфейсом, не нужно знать детали реализации. Это хорошее решение, потому что оно позволяет выполнить собственную запись

в сервисную шину. Но есть проблема. Как тестировать `AutoBrake` в изоляции? При попытке использовать производственную шину вы ступаете на территорию интеграционного тестирования и возникает необходимость в простых в настройке изолированных юнит-тестах.

Имитация зависимостей

К счастью, вы зависите не от реализации, а от интерфейса. Можно создать имитационный класс, который реализует интерфейс `IServiceBus`, и использовать его в `AutoBrake`. Имитация — это специальная реализация, которая генерируется для явной цели тестирования класса, который зависит от имитации.

Теперь при использовании `AutoBrake` в юнит-тестах он взаимодействует с имитацией, а не с производственной сервисной шиной. Поскольку вы имеете полный контроль над реализацией имитации, а имитация является классом, специфичным для юнит-теста, появляется большая гибкость в том, как можно тестировать классы, которые зависят от интерфейса:

- можно получить произвольную подробную информацию о том, как вызывается имитация. Это, например, может включать в себя информацию о параметрах и числе раз, когда имитация была вызвана;
- можно выполнять произвольные вычисления в имитации.

Другими словами, вам предоставляется полный контроль над входами и выходами зависимости `AutoBrake`. Как `AutoBrake` обрабатывает случай, когда служебная шина генерирует исключение нехватки памяти внутри вызова `publish`? Это можно протестировать. Сколько раз `AutoBrake` регистрировал обратный вызов для `SpeedUpdates`? Опять же и это можно проверить.

В листинге 10.21 представлен простой класс имитации, который можно использовать для юнит-тестов.

Листинг 10.21. Определение MockServiceBus

```
struct MockServiceBus : IServiceBus {
  void publish(const BrakeCommand& cmd) override {
    commands_published++; ❶
    last_command = cmd; ❷
  }
  void subscribe(SpeedUpdateCallback callback) override {
    speed_update_callback = callback; ❸
  }
  void subscribe(CarDetectedCallback callback) override {
    car_detected_callback = callback; ❹
  }
  BrakeCommand last_command{};
  int commands_published{};
  SpeedUpdateCallback speed_update_callback{};
  CarDetectedCallback car_detected_callback{};
};
```

Метод `publish` записывает количество раз, когда `BrakeCommand` был опубликован ❶, и `last_command`, который был опубликован ❷. Каждый раз, когда `AutoBrake` публикует команду на служебной шине, вы будете видеть обновления для членов `MockServiceBus`. Вскоре вы увидите, что это позволяет сделать несколько очень убедительных утверждений о том, как `AutoBrake` ведет себя во время теста. Здесь сохраняются функции обратного вызова, используемые для подписки на служебную шину ❸❹. Это позволяет моделировать события, вручную вызывая эти обратные вызовы для объекта имитации.

Теперь можно сосредоточиться на рефакторинге `AutoBrake`.

Рефакторинг AutoBrake

Листинг 10.22 обновляет `AutoBrake` с минимальными изменениями, необходимыми для повторной компиляции двоичного кода юнит-теста (но не обязательно проходящего!).

Листинг 10.22. Рефакторинг каркаса AutoBrake со ссылкой на IServiceBus

```
#include <exception>
--пропуск--
struct AutoBrake { ❶
  AutoBrake(IServiceBus& bus) ❷
    : collision_threshold_s{ 5 },
      speed_mps{} {
  }
  void set_collision_threshold_s(double x) {
    if (x < 1) throw std::exception{ "Collision less than 1." };
    collision_threshold_s = x;
  }
  double get_collision_threshold_s() const {
    return collision_threshold_s;
  }
  double get_speed_mps() const {
    return speed_mps;
  }
private:
  double collision_threshold_s;
  double speed_mps;
};
```

Обратите внимание, что все функции `observe` были удалены. Дополнительно `AutoBrake` больше не является шаблоном ❶. Скорее он принимает ссылку `IServiceBus` в своем конструкторе ❷.

Также нужно будет обновить юнит-тесты, чтобы снова скомпилировать набор тестов. Один из подходов, основанных на TDD, — закомментировать все тесты, которые не компилируются, и обновить `AutoBrake`, чтобы все неуспешные юнит-тесты прошли. Затем один за другим раскомментируйте каждый юнит-тест. Все они переопределяются, используя новый макет `IServiceBus`, затем обновляется `AutoBrake`, чтобы тесты проходили.

Давайте попробуем.

Рефакторинг юнит-тестов

Поскольку способ создания объекта `AutoBrake` был изменен, потребуется переопределять каждый тест. Первые три просты: в листинге 10.23 приведен пример использования конструктора `AutoBrake`.

Листинг 10.23. Повторно реализованные функции юнит-теста с использованием MockServiceBus

```
void initial_speed_is_zero() {
  MockServiceBus bus{}; ❶
  AutoBrake auto_brake{ bus }; ❷
  assert_that(auto_brake.get_speed_mps() == 0L, "speed not equal 0");
}

void initial_sensitivity_is_five() {
  MockServiceBus bus{}; ❶
  AutoBrake auto_brake{ bus }; ❷
  assert_that(auto_brake.get_collision_threshold_s() == 5,
              "sensitivity is not 5");
}

void sensitivity_greater_than_1() {
  MockServiceBus bus{}; ❶
  AutoBrake auto_brake{ bus };❷
  try {
    auto_brake.set_collision_threshold_s(0.5L);
  } catch (const std::exception&) {
    return;
  }
  assert_that(false, "no exception thrown");
}
```

Поскольку эти три теста касаются функциональности, не связанной со служебной шиной, неудивительно, что не нужно было вносить каких-либо серьезных изменений в `AutoBrake`. Все, что нужно сделать, — это создать `MockServiceBus` ❶ и передать его в конструктор `AutoBrake` ❷. Запустив пакет юнит-тестов, вы получите следующее:

```
[+] Test initial speed is 0 successful.
[+] Test initial sensitivity is 5 successful.
[+] Test sensitivity greater than 1 successful.
```

Далее посмотрите на тест `speed_is_saved`. Класс `AutoBrake` больше не предоставляет функцию `observe`, но поскольку `SpeedUpdateCallback` был сохранен на имитированной служебной шине, можно вызвать обратный вызов напрямую. Если `AutoBrake` подписан правильно, этот обратный вызов обновит скорость автомобиля и вы увидите эффект при вызове метода `get_speed_mps`. Листинг 10.24 содержит рефакторинг.

Листинг 10.24. Повторно реализованная функция юнит-теста speed_is_saved с использованием MockServiceBus

```
void speed_is_saved() {
  MockServiceBus bus{};
  AutoBrake auto_brake{ bus };

  bus.speed_update_callback(SpeedUpdate{ 100L }); ❶
  assert_that(100L == auto_brake.get_speed_mps(), "speed not saved to 100"); ❷
  bus.speed_update_callback(SpeedUpdate{ 50L });
  assert_that(50L == auto_brake.get_speed_mps(), "speed not saved to 50");
  bus.speed_update_callback(SpeedUpdate{ 0L });
  assert_that(0L == auto_brake.get_speed_mps(), "speed not saved to 0");
}
```

Тест не сильно изменился по сравнению с предыдущей реализацией. Здесь вызывается функция `speed_update_callback`, хранящаяся в имитированной шине ❶. Вы убедитесь, что объект `AutoBrake` правильно обновил скорость автомобиля ❷. Компиляция и запуск результирующего пакета юнит-тестов приводит к следующему выводу:

```
[+] Test initial speed is 0 successful.
[+] Test initial sensitivity is 5 successful.
[+] Test sensitivity greater than 1 successful.
[-] Test failure in speed is saved. bad function call.
```

Напомним, что сообщение `bad function call` происходит из исключения `std::bad_function_call`. Это ожидаемо: все еще нужно подписаться из `AutoBrake`, поэтому `std::function` выдает исключение при его вызове.

Рассмотрим подход в листинге 10.25.

Листинг 10.25. Подписка AutoBrake на обновления скорости от IServiceBus

```
struct AutoBrake {
  AutoBrake(IServiceBus& bus)
    : collision_threshold_s{ 5 },
    speed_mps{} {
    bus.subscribe([this](const SpeedUpdate& update) {
      speed_mps = update.velocity_mps;
    });
  }
  --пропуск--
}
```

Благодаря `std::function` можно передать обратный вызов в метод подписки `bus` как лямбда-выражение, которое захватывает `speed_mps`. (Обратите внимание, что не нужно сохранять копию шины.) Перекомпиляция и запуск пакета юнит-тестов дают следующее:

```
[+] Test initial speed is 0 successful.
[+] Test initial sensitivity is 5 successful.
[+] Test sensitivity greater than 1 successful.
[+] Test speed is saved successful.
```

Затем появляется первое из связанных с юнит-тестами предупреждений, `no_alert_when_not_imminent`. В листинге 10.26 показан один из способов обновления этого теста новой архитектурой.

Листинг 10.26. Обновление теста no_alert_when_not_imminent с помощью IServiceBus

```
void no_alert_when_not_imminent() {
  MockServiceBus bus{};
  AutoBrake auto_brake{ bus };
  auto_brake.set_collision_threshold_s(2L);
  bus.speed_update_callback(SpeedUpdate{ 100L }); ❶
  bus.car_detected_callback(CarDetected{ 1000L, 50L }); ❷
  assert_that(bus.commands_published == 0, "brake commands were published");
}
```

Как и в тесте `speed_is_saved`, обратные вызовы вызываются на имитированной шине для имитации событий в служебной шине ❶ ❷. Перекомпиляция и запуск пакета юнит-тестов приводит к ожидаемому сбою.

```
[+] Test initial speed is 0 successful.
[+] Test initial sensitivity is 5 successful.
[+] Test sensitivity greater than 1 successful.
[+] Test speed is saved successful.
[-] Test failure in no alert when not imminent. bad function call.
```

Необходимо подписаться на `CarDetectedCallback`. Это можно добавить в конструктор `AutoBrake`, как показано в листинге 10.27.

Листинг 10.27. Обновленный конструктор AutoBrake, который подключается к служебной шине

```
struct AutoBrake {
  AutoBrake(IServiceBus& bus)
    : collision_threshold_s{ 5 },
    speed_mps{} {
    bus.subscribe([this](const SpeedUpdate& update) {
      speed_mps = update.velocity_mps;
    });
    bus.subscribe([this❶, &bus❷](const CarDetected& cd) {
      const auto relative_velocity_mps = speed_mps - cd.velocity_mps;
      const auto time_to_collision_s = cd.distance_m / relative_velocity_mps;
      if (time_to_collision_s > 0 &&
          time_to_collision_s <= collision_threshold_s) {
        bus.publish(BrakeCommand{ time_to_collision_s }); ❸
      }
    });
  }
  --пропуск--
}
```

Все, что вы сделали, — перенесли исходный метод `observe`, соответствующий событиям `CarDetected`. Лямбда захватывает `this` ❶ и `bus` ❷ по ссылке в обратном вы-

зове. С его помощью можно вычислить время столкновения, тогда как с помощью захвата bus можно опубликовать BrakeCommand ❸, если условия выполняются. Теперь двоичный код юнит-теста выдает следующее:

```
[+] Test initial speed is 0 successful.
[+] Test initial sensitivity is 5 successful.
[+] Test sensitivity greater than 1 successful.
[+] Test speed is saved successful.
[+] Test no alert when not imminent successful.
```

Наконец, добавьте последний тест alert_when_imminent, как показано в листинге 10.28.

Листинг 10.28. Рефакторинг юнит-теста alert_when_imminent

```
void alert_when_imminent() {
  MockServiceBus bus{};
  AutoBrake auto_brake{ bus };
  auto_brake.set_collision_threshold_s(10L);
  bus.speed_update_callback(SpeedUpdate{ 100L });
  bus.car_detected_callback(CarDetected{ 100L, 0L });
  assert_that(bus.commands_published == 1, "1 brake command was not published");
  assert_that(bus.last_command.time_to_collision_s == 1L,
              "time to collision not computed correctly."); ❶
}
```

В MockServiceBus фактически сохранена последняя команда BrakeCommand, опубликованная в члене шины. В тесте можно использовать этот элемент, чтобы убедиться, что время столкновения было рассчитано правильно. Если автомобиль движется со скоростью 100 метров в секунду, нужна одна секунда, чтобы врезаться в стоящую машину, припаркованную в 100 метрах. Вы проверяете, что в BrakeCommand записано правильное время столкновения, обратившись к полю time_to_collision_s в нашей имитированной шине ❶.

Перекомпилировав и перезапустив тесты, вы наконец получите полностью зеленый набор тестов:

```
[+] Test initial speed is 0 successful.
[+] Test initial sensitivity is 5 successful.
[+] Test sensitivity greater than 1 successful.
[+] Test speed is saved successful.
[+] Test no alert when not imminent successful.
[+] Test alert when imminent successful.
```

Рефакторинг завершен.

Переоценка решения для юнит-тестирования

Оглядываясь назад на решение для юнит-тестирования, можно выделить несколько компонентов, которые не имеют ничего общего с AutoBrake. Это компоненты юнит-тестирования общего назначения, которые можно использовать

в будущих юнит-тестах. Вспомните две вспомогательные функции, созданные в листинге 10.29.

Листинг 10.29. Строгая структура юнит-тестирования

```
#include <stdexcept>
#include <cstdio>

void assert_that(bool statement, const char* message) {
  if (!statement) throw std::runtime_error{ message };
}

void run_test(void(*unit_test)(), const char* name) {
  try {
    unit_test();
    printf("[+] Test %s successful.\n", name);
    return;
  } catch (const std::exception& e) {
    printf("[-] Test failure in %s. %s.\n", name, e.what());
  }
}
```

Эти две функции отражают два фундаментальных аспекта юнит-тестирования: создание утверждений и выполнение тестов. Запуск своих собственных простых функций `assert_that` и `run_test` сработает, но этот подход не очень хорошо масштабируется. Можно сделать это намного лучше, опираясь на фреймворк юнит-тестирования.

Фреймворки юнит-тестирования и имитации

Фреймворки юнит-тестирования предоставляют часто используемые функции и основы, которые нужны, чтобы связать тесты в удобную для пользователя программу. Эти фреймворки предоставляют множество функциональных возможностей, которые помогают создавать краткие выразительные тесты. В этом разделе мы рассмотрим несколько популярных фреймворков для юнит-тестирования и мокирования (имитации).

Фреймворк юнит-тестирования Catch

Один из самых простых фреймворков юнит-тестирования, Catch, созданный Филом Нэшем (Phil Nash), доступен по адресу github.com/catchorg/Catch2/. Поскольку эта библиотека предназначена только для заголовков, можно настроить Catch, загрузив версию с одним заголовком (каталог single_header) и включив ее в каждую единицу трансляции, содержащую код юнит-тестирования.

ПРИМЕЧАНИЕ

На момент выхода книги последняя версия Catch — 2.9.1.

Определение точки входа

Укажите Catch предоставить точку входа тестового двоичного файла с помощью `#define CATCH_CONFIG_MAIN`. Пакет юнит-тестирования Catch запускается следующим образом:

```
#define CATCH_CONFIG_MAIN
#include "catch.hpp"
```

Вот и все. В заголовке `catch.hpp` код ищет определение препроцессора `CATCH_CONFIG_MAIN`. Если оно присутствует, Catch добавит его в функцию `main`, так что вам не нужно ничего делать самостоятельно. Он автоматически соберет все определенные юнит-тесты и обернет их в подходящее окружение кода.

Определение тестовых случаев

Ранее в разделе «Юнит-тестирование» на с. 351 мы определили отдельную функцию для каждого юнит-теста. Затем нужно передать указатель на эту функцию в качестве первого параметра для `run_test`. Имя теста было передано в качестве второго параметра, который немного избыточен, потому что вы уже предоставили описательное имя для функции, на которую указывает первый аргумент. Наконец, пришлось реализовать собственную функцию `assert`. Catch обрабатывает всю эту церемонию безоговорочно. Для каждого юнит-теста используется макрос TEST_CASE, а Catch обрабатывает всю интеграцию самостоятельно.

В листинге 10.30 показано, как создать простую программу юнит-тестирования с помощью Catch.

Листинг 10.30. Простая программа юнит-тестирования Catch

```
#define CATCH_CONFIG_MAIN
#include "catch.hpp"

TEST_CASE("AutoBrake") { ❶
  // Юнит-тест
}
-------------------------------------------------------------------------------
test cases: 1 | 1 passed ❶
assertions: - none - ❷
```

Точка входа Catch обнаруживает, что был объявлен один тест с именем `AutoBrake` ❶. Она также предупреждает, что вы не предоставили никаких утверждений ❷.

Создание утверждений

Catch предоставляется со встроенным утверждением, которое имеет два разных семейства макросов утверждений: `REQUIRE` и `CHECK`. Разница между ними в том, что `REQUIRE` немедленно провалит тест, в то время как `CHECK` позволит выполнить тест до его завершения (но все равно вызовет сбой). `CHECK` может быть полезен иногда, когда группы связанных утверждений, которые проходят неуспешно, ведут программиста по правильному пути устранения проблем отладки. Также включены `REQUIRE_FALSE`

и CHECK_FALSE, которые проверяют, что содержащийся в них оператор оценивается как ложный, а не как истинный. В некоторых ситуациях может показаться, что это более естественный способ представления требования.

Все, что нужно сделать, — это обернуть логическое выражение макросом REQUIRE. Если выражение оценивается как ложное, утверждение не выполняется. Вы предоставляете выражение подтверждения, которое вычисляется как true, если утверждение проходит, и как false, если оно не выполняется:

```
REQUIRE(выражение подтверждения);
```

Давайте посмотрим, как объединить REQUIRE с TEST_CASE для создания юнит-теста.

ПРИМЕЧАНИЕ

Поскольку это наиболее распространенное утверждение Catch, мы будем использовать здесь REQUIRE. Обратитесь к документации Catch для получения дополнительной информации.

Рефакторинг initial_speed_is_zero теста для Catch

В листинге 10.31 показан тест initial_speed_is_zero, реорганизованный для использования Catch.

Листинг 10.31. Юнит-тест initial_speed_is_zero с рефакторингом для использования Catch

```
#define CATCH_CONFIG_MAIN
#include "catch.hpp"
#include <functional>

struct IServiceBus {
  --пропуск--
};

struct MockServiceBus : IServiceBus {
  --пропуск--
};

struct AutoBrake {
  --пропуск--
};

TEST_CASE❶("initial car speed is zero"❷) {
  MockServiceBus bus{};
  AutoBrake auto_brake{ bus };
  REQUIRE(auto_brake.get_speed_mps() == 0); ❸
}
```

Макрос TEST_CASE используется для определения нового юнит-теста ❶. Тест описывается единственным параметром ❷. Внутри тела макроса TEST_CASE юнит-тест продолжается. Вы также видите макрос REQUIRE в действии ❸. Чтобы увидеть, как Catch обрабатывает неудачные тесты, закомментируйте инициализатор элемента

speed_mps, чтобы вызвать провал теста, и просмотрите выходные данные программы, как показано в листинге 10.32.

Листинг 10.32. Намеренное закомментирование инициализатора элемента speed_mps для провала теста (с использованием Catch)

```
struct AutoBrake {
  AutoBrake(IServiceBus& bus)
    : collision_threshold_s{ 5 }/*,
    speed_mps{} */{ ❶
  --пропуск--
};
```

Соответствующий инициализатор члена ❶ закомментирован, что приводит к провалу теста. Повторный запуск набора тестов Catch из листинга 10.31 приводит к выводу, отраженному в листинге 10.33.

Листинг 10.33. Результат запуска набора тестов после реализации листинга 10.31.

```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
catch_example.exe is a Catch v2.0.1 host application.
Run with -? for options

-------------------------------------------------------------------------------
initial car speed is zero
-------------------------------------------------------------------------------
c:\users\jalospinoso\catch-test\main.cpp(82)
...............................................................................

c:\users\jalospinoso\catch-test\main.cpp(85):❶ FAILED:
  REQUIRE( auto_brake.get_speed_mps()L == 0 ) ❷
with expansion:
  -92559631349317830736831783200707727132248687965119994463780864.0 ❸
  ==
  0
===============================================================================
test cases: 1 | 1 failed
assertions: 1 | 1 failed
```

Такие результаты в разы лучше полученных в первоначальном юнит-тесте. Catch сообщает точную строку, в которой юнит-тест не прошел ❶, а затем самостоятельно выводит эту строку ❷. Затем он расширяет эту строку до фактических значений, обнаруженных во время выполнения. Вы можете заметить, что гротескное (неинициализированное) значение, возвращаемое get_speed_mps(), явно не равно 0 ❸. Сравните эти выходные данные с результатами доменного теста. Я думаю, вы согласитесь, что использование Catch имеет настоящую ценность.

Утверждения и исключения

Catch также предоставляет специальное утверждение REQUIRE_THROWS. Этот макрос требует, чтобы содержащееся в нем выражение вызывало исключение. Для достижения аналогичной функциональности в первоначальном юнит-тесте рассмотрим это многострочное чудовище:

```
try {
  auto_brake.set_collision_threshold_s(0.5L);
} catch (const std::exception&) {
  return;
}
assert_that(false, "no exception thrown");
```

Также доступны другие макросы, поддерживающие исключения. Можно потребовать, чтобы некоторые выражения не генерировали исключение, используя макросы `REQUIRE_NOTHROW` и `CHECK_NOTHROW`. Также можно точно указать ожидаемый тип исключения, используя макросы `REQUIRE_THROWS_AS` и `CHECK_THROWS_AS`. Они ожидают второй параметр, описывающий ожидаемый тип. Их использование похоже на `REQUIRE`; вы просто предоставляете какое-то выражение, которое должно выдать исключение для подтверждения утверждения:

```
REQUIRE_THROWS(выражение-для-вычисления);
```

Если *выражение-для-вычисления* не выдает исключение, утверждение не выполняется.

Утверждения с плавающей точкой

Класс `AutoBrake` включает в себя арифметику с плавающей точкой, и до этого потенциально серьезная проблема утверждений не затрагивалась. Поскольку числа с плавающей точкой влекут за собой ошибки округления, проверять равенство с помощью оператора `==` не очень хорошая идея. Более надежный подход заключается в проверке того, является ли разница между числами с плавающей запятой сколь угодно малой. С помощью Catch можно легко справиться с этими ситуациями, используя класс `Approx`, как показано в листинге 10.34.

Листинг 10.34. Рефакторинг теста «инициализирует чувствительность к пяти» с использованием класса Approx

```
TEST_CASE("AutoBrake") {
  MockServiceBus bus{};
  AutoBrake auto_brake{ bus };
  REQUIRE(auto_brake.get_collision_threshold_s() == Approx(5L));
}
```

Класс `Approx` помогает Catch выполнять допустимые сравнения значений с плавающей точкой. Он может существовать с любой стороны выражения сравнения. Он имеет разумные значения по умолчанию для определения толерантности, но вы имеете полный контроль над особенностями (см. документацию Catch по `epsilon`, `margin` и `scale`).

Сбой

Можно вызвать сбой теста Catch с помощью макроса `FAIL()`. Иногда это может быть полезно в сочетании с условными выражениями, например:

```
if (something-bad) FAIL("что-то-не-так.")
```

Используйте оператор `REQUIRE`, если он является подходящим.

Тестовые случаи и разделы

Catch поддерживает идею тестовых случаев и разделов, которые значительно упрощают общую настройку и разборку юнит-тестов. Обратите внимание, что каждый из тестов повторяется каждый раз при создании `AutoBrake`:

```
MockServiceBus bus{};
AutoBrake auto_brake{ bus };
```

Нет необходимости повторять этот код снова и снова. Решением Catch для этой распространенной установки является использование вложенных макросов `SECTION`. Можно вкладывать макросы `SECTION` в `TEST_CASE` в основной шаблон использования, как показано в листинге 10.35.

Листинг 10.35. Пример настройки Catch с вложенными макросами

```
TEST_CASE("MyTestGroup") {
  // Код настройки ❶
  SECTION("MyTestA") { ❷
    // Код для теста A
  }
  SECTION("MyTestB") { ❸
    // Код для теста B
  }
}
```

Все настройки можно выполнить один раз в начале `TEST_CASE` ❶. Когда Catch видит макросы `SECTION`, вложенные в `TEST_CASE`, он (концептуально) копирует и вставляет все настройки в каждый `SECTION` ❷❸. Каждый `SECTION` выполняется независимо от других, поэтому обычно любые побочные эффекты для объектов, созданных в `TEST_CASE`, не наблюдаются в макросах `SECTION`. Кроме того, можно встроить макрос `SECTION` в другой макрос `SECTION`. Это может быть полезно, если у вас есть много установочного кода для набора тесно связанных тестов (хотя может иметь смысл разделить этот набор на его собственный `TEST_CASE`).

Давайте посмотрим, как этот подход упрощает пакет юнит-тестов `AutoBrake`.

Рефакторинг юнит-тестов AutoBrake с Catch

Листинг 10.36 переводит все юнит-тесты в стиль Catch.

Листинг 10.36. Использование фреймворка Catch для реализации юнит-тестов

```
#define CATCH_CONFIG_MAIN
#include "catch.hpp"
#include <functional>
#include <stdexcept>

struct IServiceBus {
  --пропуск--
};

struct MockServiceBus : IServiceBus {
```

```
  --пропуск--
};

struct AutoBrake {
  --пропуск--
};

TEST_CASE("AutoBrake"❶) {
  MockServiceBus bus{}; ❷
  AutoBrake auto_brake{ bus }; ❸

  SECTION❹("initializes speed to zero"❺) {
    REQUIRE(auto_brake.get_speed_mps() == Approx(0));
  }

  SECTION("initializes sensitivity to five") {
    REQUIRE(auto_brake.get_collision_threshold_s() == Approx(5));
  }

  SECTION("throws when sensitivity less than one") {
    REQUIRE_THROWS(auto_brake.set_collision_threshold_s(0.5L));
  }

  SECTION("saves speed after update") {
    bus.speed_update_callback(SpeedUpdate{ 100L });
    REQUIRE(100L == auto_brake.get_speed_mps());
    bus.speed_update_callback(SpeedUpdate{ 50L });
    REQUIRE(50L == auto_brake.get_speed_mps());
    bus.speed_update_callback(SpeedUpdate{ 0L });
    REQUIRE(0L == auto_brake.get_speed_mps());
  }

  SECTION("no alert when not imminent") {
    auto_brake.set_collision_threshold_s(2L);
    bus.speed_update_callback(SpeedUpdate{ 100L });
    bus.car_detected_callback(CarDetected{ 1000L, 50L });
    REQUIRE(bus.commands_published == 0);
  }

  SECTION("alert when imminent") {
    auto_brake.set_collision_threshold_s(10L);
    bus.speed_update_callback(SpeedUpdate{ 100L });
    bus.car_detected_callback(CarDetected{ 100L, 0L });
    REQUIRE(bus.commands_published == 1);
    REQUIRE(bus.last_command.time_to_collision_s == Approx(1));
  }
}
-------------------------------------------------------------------------------
===============================================================================
All tests passed (9 assertions in 1 test case)
```

Здесь TEST_CASE переименовывается в AutoBrake, чтобы отразить его более общее назначение ❶. Затем тело TEST_CASE начинается с общего кода настройки, который разделяется всеми юнит-тестами AutoBrake ❷❸. Каждый из юнит-тестов был преобразован в макрос SECTION ❹. Каждому из разделов присваивается имя ❺, и затем специфичный для теста код помещается в тело SECTION. Catch выполнит всю работу по соединению кода настройки с каждым из тел SECTION. Другими словами, каждый раз получается новый AutoBrake. Порядок SECTION здесь не имеет значения, и они полностью независимы.

Google Test

Google Test — еще одна чрезвычайно популярная платформа для юнит-тестирования. GoogleTest придерживается традиций платформы юнит-тестирования xUnit, и если вы знакомы, например, с junit для Java или nunit для .NET, то будете чувствовать себя как дома, используя Google Test. Приятной особенностью использования Google Test является то, что некоторое время назад он был объединен с фреймворком имитации Google Mocks.

Настройка Google Test

Предварительная настройка Google Test занимает какое-то время. В отличие от Catch, GoogleTest не является библиотекой только для заголовков. Его нужно загрузить с github.com/google/googletest/, скомпилировать в набор библиотек и при необходимости связать эти библиотеки с тестовым проектом. Если вы используете популярную систему сборки для ПК вроде GNU Make, Apple Xcode или Visual Studio, доступны некоторые шаблоны, которые можно использовать для начала создания соответствующих библиотек.

Для получения дополнительной информации об установке и запуске Google Test обратитесь к руководству, доступному в каталоге docs репозитория.

ПРИМЕЧАНИЕ

На момент выхода книги последняя версия Google Test — 1.8.1. Перейдите по адресу ccc.codes, чтобы узнать о методе интеграции Google Test в сборку Cmake.

В рамках проекта юнит-тестирования нужно выполнить две операции для настройки Google Test. Во-первых, следует убедиться, что включенный каталог установки Google Test находится в пути поиска заголовка проекта. Это позволяет использовать `#include "gtest/gtest.h"` в тестах. Во-вторых, нужно указать редактору связей включить статические библиотеки gtest и gtest_main из установки Google Test. Убедитесь, что вы указали правильную архитектуру и параметры конфигурации вашего компьютера.

ПРИМЕЧАНИЕ

Обычная ошибка при настройке Google Test в Visual Studio состоит в том, что опция C/C++->Code Generation -> Runtime Library в Google Test должна соответствовать параметру вашего проекта. По умолчанию Google Test компилирует среду выполнения статически (то есть с параметрами /MT или /MTd). Этот выбор отличается от значения по умолчанию, которое заключается в динамической компиляции среды выполнения (например, с помощью параметров /MD или /MDd в Visual Studio).

Определение точки входа

Google Test предоставит функцию `main()` после добавления `gtest_main` в проект юнит-тестирования. Представьте это как аналог Google Test для `#defineCATCH_CONFIG_MAIN` из Catch; он найдет все заданные юнит-тесты и соберет их вместе в красивое тестовое окружение.

Определение тестовых случаев

Чтобы определить тестовые случаи, нужно предоставить юнит-тесты с использованием макроса `TEST`, который очень похож на `TEST_CASE` в Catch. В листинге 10.37 показана основная настройка юнит-теста Google Test.

Листинг 10.37. Пример юнит-теста Google Test

```
#include "gtest/gtest.h" ❶

TEST❷(AutoBrake❸, UnitTestName❹) {
  // Юнит-тест ❺
}
-------------------------------------------------------------------------------
Running main() from gtest_main.cc ❻
[==========] Running 1 test from 1 test case.
[----------] Global test environment set-up.
[----------] 1 test from AutoBrake
[ RUN      ] AutoBrake.UnitTestName
[       OK ] AutoBrake.UnitTestName (0 ms)
[----------] 1 test from AutoBrake (0 ms total)

[----------] Global test environment tear-down
[==========] 1 test from 1 test case ran. (1 ms total)
[  PASSED  ] 1 test. ❼
```

Сначала добавляется заголовок `gtest/gtest.h` ❶. Он включает в себя все определения, необходимые для определения юнит-тестов. Каждый юнит-тест начинается с макроса `TEST` ❷. Каждый юнит-тест определяется двумя метками: *именем тестового случая* — в нашем случае `AutoBrake` ❸, и *именем теста* — `UnitTestName` ❹. Они примерно аналогичны именам `TEST_CASE` и `SECTION` (соответственно) в Catch. Тестовый набор содержит один или несколько тестов. Обычно тесты, которые имеют общую тему, размещаются вместе. Фреймворк сгруппирует тесты, что может быть полезно для некоторых более продвинутых целей. Различные тестовые случаи могут иметь тесты с одинаковым именем.

Код юнит-теста нужно поместить в фигурные скобки ❺. При запуске получившегося двоичного файла юнит-теста вы увидите, что Google Test предоставляет точку входа ❻. Поскольку не было предоставлено никаких утверждений (или кода, который может вызвать исключение), юнит-тесты проходят ❼.

Создание утверждений

Утверждения в Google Test менее волшебны, чем в `REQUIRE` в Catch. Хотя они также являются макросами, утверждения Google Test требуют гораздо большей работы со стороны программиста. В тех случаях, когда `REQUIRE` будет анализировать логическое выражение и определять, проверяется ли равенство, отношение «больше» и т. д., утверждения Google Test этого не сделают. Нужно передать каждый компонент утверждения отдельно.

Есть много других вариантов формулировки утверждений в Google Test. Они отображаются в табл. 10.1.

Таблица 10.1. Утверждения Google Test

Утверждение	Проверяет, что
ASSERT_TRUE(*условие*)	*условие* равно true
ASSERT_FALSE(*условие*)	*условие* равно false
ASSERT_EQ(*арг1*, *арг2*)	*арг1* == *арг2* равно true
ASSERT_FLOAT_EQ(*арг1*, *арг2*)	*арг1* — *арг2* равно погрешности округления (float)
ASSERT_DOUBLE_EQ(*арг1*, *арг2*)	*арг1* — *арг2* равно погрешности округления (double)
ASSERT_NE(*арг1*, *арг2*)	*арг1* != *арг2* равно true
ASSERT_LT(*арг1*, *арг2*)	*арг1* < *арг2* равно true
ASSERT_LE(*арг1*, *арг2*)	*арг1* <= *арг2* равно true
ASSERT_GT(*арг1*, *арг2*)	*арг1* > *арг2* равно true
ASSERT_GE(*арг1*, *арг2*)	*арг1* >= *арг2* равно true
ASSERT_STREQ(*стр1*, *стр2*)	Две строки в стиле C, *стр1* и *стр2*, имеют одинаковое содержание
ASSERT_STRNE(*стр1*, *стр2*)	Две строки в стиле C, *стр1* и *стр2*, имеют разное содержание
ASSERT_STRCASEEQ(*стр1*, *стр2*)	Две строки в стиле C, *стр1* и *стр2*, имеют одинаковое содержание, за исключением регистра
ASSERT_STRCASENE(*стр1*, *стр2*)	Две строки в стиле C, *стр1* и *стр2*, имеют разное содержание, за исключением регистра
ASSERT_THROW(*выражение*, *тип-исключения*)	Вычисление *выражения* приведет к выбрасыванию исключения с типом *тип-исключения*

Продолжение ⇩

Таблица 10.1 *(продолжение)*

Утверждение	Проверяет, что
ASSERT_ANY_THROW(*выражение*)	Вычисление *выражения* приведет к выбрасыванию исключения любого типа
ASSERT_NO_THROW(*выражение*)	Вычисление *выражения* не приведет к выбрасыванию исключения
ASSERT_HRESULT_SUCCEEDED (*выражение*)	HRESULT, возвращаемый *выражением*, соответствует успеху (только в API Win32)
ASSERT_HRESULT_FAILED (*выражение*)	HRESULT, возвращаемый *выражением*, соответствует провалу (только в API Win32)

Давайте объединим определение юнит-теста с утверждением, чтобы увидеть Google Test в действии.

Рефакторинг теста initial_car_speed_is_zero в Google Test

С преднамеренно поврежденным `AutoBrake` в листинге 10.32 можно запустить следующий юнит-тест, чтобы увидеть, как выглядят сообщения о сбоях тестового окружения. (Вспомните, что инициализатор члена для `speed_mps` закомментирован.) В листинге 10.38 используется `ASSERT_FLOAT_EQ`, чтобы утверждать, что начальная скорость автомобиля равна нулю.

Листинг 10.38. Преднамеренное комментирование инициализатора элемента speed_mps для провала теста (с помощью Google Test)

```
#include "gtest/gtest.h"
#include <functional>

struct IServiceBus {
  --пропуск--
};

struct MockServiceBus : IServiceBus {
  --пропуск--
};

struct AutoBrake {
  AutoBrake(IServiceBus& bus)
    : collision_threshold_s{ 5 }/*,
    speed_mps{} */ {
  --пропуск--
};

TEST❶(AutoBrakeTest❷, InitialCarSpeedIsZero❸) {
  MockServiceBus bus{};
  AutoBrake auto_brake{ bus };
  ASSERT_FLOAT_EQ❹(0❺, auto_brake.get_speed_mps()❻);
}
---------------------------------------------------------------------------
```

```
Running main() from gtest_main.cc
[==========] Running 1 test from 1 test case.
[----------] Global test environment set-up.
[----------] 1 test from AutoBrakeTest
[ RUN      ] AutoBrakeTest.InitialCarSpeedIsZero
C:\Users\josh\AutoBrake\gtest.cpp(80): error: Expected equality of these
values:
  0 ❺
  auto_brake.get_speed_mps()❻
    Which is: -inf
[  FAILED  ] AutoBrakeTest❷.InitialCarSpeedIsZero❸ (5 ms)
[----------] 1 test from AutoBrakeTest (5 ms total)

[----------] Global test environment tear-down
[==========] 1 test from 1 test case ran. (7 ms total)
[  PASSED  ] 0 tests.
[  FAILED  ] 1 test, listed below:
[  FAILED  ] AutoBrakeTest.InitialCarSpeedIsZero
1 FAILED TEST
```

Объявляется юнит-тест ❶ с именем тестового случая `AutoBrakeTest` ❷ и именем теста `InitialCarSpeedIsZero` ❸. В рамках теста устанавливается `auto_brake` и утверждается ❹, что начальная скорость автомобиля равна нулю ❺. Обратите внимание, что постоянное значение является первым параметром, а проверяемое количество — вторым параметром ❻.

Как и вывод Catch в листинге 10.33, вывод Google Test в листинге 10.38 очень ясен. Он сообщает, что тест не пройден, идентифицирует ошибочное утверждение и дает хорошее представление о том, как можно решить проблему.

Испытательные приспособления

В отличие от подходов `TEST_CASE` и `SECTION` в Catch, подход Google Test заключается в формулировании *классов тестовых приспособлений* при использовании общей настройки. Эти приспособления являются классами, которые наследуются от класса`::testing::Test`, предоставляемого платформой.

Любые члены, которые предполагается использовать внутри тестов, должны быть помечены как публичные или приватные. Если нужны вычисления для установки или разборки, можно поместить их в конструктор (по умолчанию) или деструктор (соответственно).

ПРИМЕЧАНИЕ

Также можно разместить такую логику настройки и разборки в переопределенных `SetUp()` и `TearDown()` функциях, хотя это редко случается. Один из случаев — если вычисление разборки может вызвать исключение. Поскольку, как правило, не стоит позволять генерировать необработанное исключение из деструктора, придется помещать такой код в функцию `TearDown()`. (Вспомните из «Выбрасывания исключений из деструктора» на с. 166, что выбрасывание необработанного исключения в деструкторе, когда другое исключение уже находится в полете, вызывает `std::terminate`.)

Если тестовое приспособление похоже на `TEST_CASE` в Catch, то `TEST_F` похож на `SECTION` в Catch. Как и `TEST`, `TEST_F` принимает два параметра. Первым должно быть точное название класса тестового приспособления. Второе — это название юнит-теста.

Листинг 10.39 показывает основное использование тестовых приспособлений в Google Test.

Листинг 10.39. Базовая настройка тестовых приспособлений в Google Test

```
#include "gtest/gtest.h"

struct MyTestFixture❶ : ::testing::Test❷ { };

TEST_F(MyTestFixture❸, MyTestA❹) {
  // Test A here
}

TEST_F(MyTestFixture, MyTestB❺) {
  // Test B here
}
-------------------------------------------------------------------------------
Running main() from gtest_main.cc
[==========] Running 2 tests from 1 test case.
[----------] Global test environment set-up.
[----------] 2 tests from MyTestFixture
[ RUN      ] MyTestFixture.MyTestA
[       OK ] MyTestFixture.MyTestA (0 ms)
[ RUN      ] MyTestFixture.MyTestB
[       OK ] MyTestFixture.MyTestB (0 ms)
[----------] 2 tests from MyTestFixture (1 ms total)

[----------] Global test environment tear-down
[==========] 2 tests from 1 test case ran. (3 ms total)
[  PASSED  ] 2 tests.
```

Объявляется класс `MyTestFixture` ❶, который наследуется от класса `::testing::Test`, предоставляемого Google Test ❷. Имя класса используется в качестве первого параметра макроса `TEST_F` ❸. Затем юнит-тест получает доступ к любым публичным или приватным методам внутри `MyTestFixture` и можно использовать конструктор и деструктор `MyTestFixture` для выполнения любой обычной установки/разборки теста. Второй аргумент — это имя юнит-теста ❹❺.

Далее давайте посмотрим, как использовать Google Test Fixtures для переопределения юнит-тестов AutoBrake.

Рефакторинг юнит-тестов AutoBrake с помощью Google Test

Листинг 10.40 переопределяет все юнит-тесты AutoBrake в вариант фреймворка тестовых приспособлений Google Test.

Листинг 10.40. Использование Google Test для реализации юнит-тестов AutoBrake

```
#include "gtest/gtest.h"
#include <functional>

struct IServiceBus {
  --пропуск--
};

struct MockServiceBus : IServiceBus {
  --пропуск--
};

struct AutoBrake {
  --пропуск--
};

struct AutoBrakeTest : ::testing::Test { ❶
  MockServiceBus bus{};
  AutoBrake auto_brake { bus };
};

TEST_F❷(AutoBrakeTest❸, InitialCarSpeedIsZero❹) {
  ASSERT_DOUBLE_EQ(0, auto_brake.get_speed_mps()); ❺
}

TEST_F(AutoBrakeTest, InitialSensitivityIsFive) {
  ASSERT_DOUBLE_EQ(5, auto_brake.get_collision_threshold_s());
}

TEST_F(AutoBrakeTest, SensitivityGreaterThanOne) {
  ASSERT_ANY_THROW(auto_brake.set_collision_threshold_s(0.5L)); ❻
}

TEST_F(AutoBrakeTest, SpeedIsSaved) {
  bus.speed_update_callback(SpeedUpdate{ 100L });
  ASSERT_EQ(100, auto_brake.get_speed_mps());
  bus.speed_update_callback(SpeedUpdate{ 50L });
  ASSERT_EQ(50, auto_brake.get_speed_mps());
  bus.speed_update_callback(SpeedUpdate{ 0L });
  ASSERT_DOUBLE_EQ(0, auto_brake.get_speed_mps());
}

TEST_F(AutoBrakeTest, NoAlertWhenNotImminent) {
  auto_brake.set_collision_threshold_s(2L);
  bus.speed_update_callback(SpeedUpdate{ 100L });
  bus.car_detected_callback(CarDetected{ 1000L, 50L });
  ASSERT_EQ(0, bus.commands_published);
}

TEST_F(AutoBrakeTest, AlertWhenImminent) {
  auto_brake.set_collision_threshold_s(10L);
  bus.speed_update_callback(SpeedUpdate{ 100L });
```

```
  bus.car_detected_callback(CarDetected{ 100L, 0L });
  ASSERT_EQ(1, bus.commands_published);
  ASSERT_DOUBLE_EQ(1L, bus.last_command.time_to_collision_s);
}
-------------------------------------------------------------------------------
Running main() from gtest_main.cc
[==========] Running 6 tests from 1 test case.
[----------] Global test environment set-up.
[----------] 6 tests from AutoBrakeTest
[ RUN      ] AutoBrakeTest.InitialCarSpeedIsZero
[       OK ] AutoBrakeTest.InitialCarSpeedIsZero (0 ms)
[ RUN      ] AutoBrakeTest.InitialSensitivityIsFive
[       OK ] AutoBrakeTest.InitialSensitivityIsFive (0 ms)
[ RUN      ] AutoBrakeTest.SensitivityGreaterThanOne
[       OK ] AutoBrakeTest.SensitivityGreaterThanOne (1 ms)
[ RUN      ] AutoBrakeTest.SpeedIsSaved
[       OK ] AutoBrakeTest.SpeedIsSaved (0 ms)
[ RUN      ] AutoBrakeTest.NoAlertWhenNotImminent
[       OK ] AutoBrakeTest.NoAlertWhenNotImminent (1 ms)
[ RUN      ] AutoBrakeTest.AlertWhenImminent
[       OK ] AutoBrakeTest.AlertWhenImminent (0 ms)
[----------] 6 tests from AutoBrakeTest (3 ms total)

[----------] Global test environment tear-down
[==========] 6 tests from 1 test case ran. (4 ms total)
[  PASSED  ] 6 tests.
```

Сначала реализуется тестовое приспособление `AutoBrakeTest` ❶. Этот класс инкапсулирует общий код установки во всех юнит-тестах: для создания `MockServiceBus` и его использования для создания `AutoBrake`. Каждый из юнит-тестов представлен макросом `TEST_F` ❷. Эти макросы принимают два параметра: тестовое приспособление, например `AutoBrakeTest` ❸, и имя теста, например `InitialCarSpeedIsZero` ❹. В теле юнит-тестов представлены правильные вызовы для каждого из утверждений, такие как `ASSERT_DOUBLE_EQ` ❺ и `ASSERT_ANY_THROW` ❻.

Сравнение Google Test и Catch

Как вы уже заметили, существует несколько важных отличий между Google Test и Catch. Во-первых, это предварительная подготовка Google Test и его корректная работа. Catch находится на противоположном конце спектра: в качестве библиотеки только для заголовков заставить работать Catch в проекте весьма просто.

Еще одним важным отличием являются утверждения. Для новичка `REQUIRE` намного проще в использовании, чем стиль утверждений в Google Test. Опытному пользователю другого фреймворка xUnit Google Test может показаться более естественным. Сообщения об ошибках также немного отличаются. Вам решать, какой из этих стилей является более подходящим.

Наконец, учитывайте производительность. Теоретически Google Test будет компилироваться быстрее, чем Catch, потому что весь Catch должен быть скомпилирован

для каждой единицы трансляции в наборе юнит-тестов. Это компромисс для библиотек только для заголовков; инвестиции в настройку, которая осуществляется при установке Google Test, окупаются позже с более быстрой компиляцией. Это может или не может быть ощутимо в зависимости от размера набора юнит-тестов.

Boost Test

Boost Test — это фреймворк юнит-тестирования, который поставляется как часть *библиотек Boost C++* (или просто *Boost*). Boost — отличная коллекция библиотек C++ с открытым исходным кодом. В его истории появлялось множество идей, которые в конечном итоге были включены в стандарт C++, хотя не все библиотеки Boost стремятся к возможному включению. Вы увидите упоминание о ряде библиотек Boost в оставшейся части этой книги, и Boost Test является первым из них. Для получения справки по установке boost в вашей среде посетите домашнюю страницу Boost www.boost.org или ознакомьтесь с сопутствующим кодом этой книги.

ПРИМЕЧАНИЕ

На момент выхода книги последняя версия библиотек Boost — 1.70.0.

Boost Test можно использовать в трех режимах: в качестве библиотеки только для заголовков (как, например, Catch), в качестве статической библиотеки (как, например, Google Test) или в качестве общей библиотеки, которая будет связывать модуль Boost Test во время выполнения. Использование динамической библиотеки может сэкономить немного места на диске, если у вас есть несколько бинарных юнит-тестов.

Вместо того чтобы вставлять среду юнит-тестирования в каждый из двоичных файлов юнит-теста, можно создать единую общую библиотеку (например, *.so* или *.dll*) и загрузить ее во время выполнения.

Как вы узнали при изучении Catch и Google Test, с каждым из этих подходов связаны компромиссы. Основным преимуществом Boost Test является то, что он позволяет выбрать лучший режим по своему усмотрению. Переключать режимы не очень сложно, если проект развивается, поэтому один из возможных подходов — начать использовать Boost Test в качестве библиотеки только для заголовков и перейти в другой режим при изменении требований.

Настройка Boost Test

Чтобы настроить Boost Test в режиме «только заголовок» (то, что в документации Boost называется «вариант с одним заголовком»), просто включите заголовок `<boost/test/included/unit_test.hpp>`. Чтобы этот заголовок компилировался, нужно определить `BOOST_TEST_MODULE` с пользовательским именем. Например:

```
#define BOOST_TEST_MODULE test_module_name
#include <boost/test/included/unit_test.hpp>
```

К сожалению, нельзя использовать этот подход, если у вас есть более одной единицы трансляции. Для таких ситуаций Boost Test содержит готовые статические библиотеки, которые можно использовать. Связывая их, вы избегаете необходимости компилировать один и тот же код для каждой единицы трансляции. При использовании этого подхода заголовок `boost/test/unit_test.hpp` включается для каждой единицы трансляции в наборе юнит-тестов:

```
#include <boost/test/unit_test.hpp>
```

Ровно в одну единицу трансляции нужно также включить определение `BOOST_TEST_MODULE`:

```
#define BOOST_TEST_MODULE AutoBrake
#include <boost/test/unit_test.hpp>
```

Также необходимо настроить редактор связей для включения соответствующей статической библиотеки Boost Test, которая поставляется с установкой Boost Test. Компилятор и архитектура, соответствующие выбранной статической библиотеке, должны соответствовать остальной части проекта юнит-тестирования.

Настройка режима совместно используемой библиотеки

Чтобы настроить Boost Test в режиме совместно используемой библиотеки, необходимо добавить следующие строки в каждую единицу трансляции набора юнит-тестов:

```
#define BOOST_TEST_DYN_LINK
#include <boost/test/unit_test.hpp>
```

Ровно в *одной* единице трансляции также необходимо определить `BOOST_TEST_MODULE`:

```
#define BOOST_TEST_MODULE AutoBrake
#define BOOST_TEST_DYN_LINK
#include <boost/test/unit_test.hpp>
```

Как и в случае использования статической библиотеки, нужно указать редактору связей включить Boost Test. Во время выполнения общая библиотека юнит-теста также должна быть доступна.

Определение тестовых случаев

Можно определить юнит-тест в Boost Test с помощью макроса `BOOST_AUTO_TEST_CASE`, который принимает один параметр, соответствующий имени теста.

Листинг 10.41 показывает основное использование.

Листинг 10.41. Использование Boost Test для реализации юнит-тестов AutoBrake

```
#define BOOST_TEST_MODULE TestModuleName ❶
#include <boost/test/unit_test.hpp> ❷

BOOST_AUTO_TEST_CASE❸(TestA❹) {
```

```
  // Юнит-тест A ❺
}
-------------------------------------------------------------------------------
Running 1 test case...

*** No errors detected
```

Имя тестового модуля — `TestModuleName` ❶, который определяется как `BOOST_TEST_MODULE`. Добавляется заголовок `boost/test/unit_test.hpp` ❷, который предоставляет доступ ко всем нужным компонентам Boost Test. Объявление `BOOST_AUTO_TEST_CASE` ❸ обозначает юнит-тест с именем `TestA` ❹. Тело юнит-теста проходит между фигурными скобками ❺.

Создание утверждений

Утверждения в Boost очень похожи на утверждения в Catch. Макрос `BOOST_TEST` похож на макрос `REQUIRE` в Catch. Нужно просто предоставить выражение, которое вычисляется как `true`, если утверждение проходит, и как `false` в обратном случае:

```
BOOST_TEST(выражение-утверждения)
```

`assertion-expression` — выражение-утверждения.

Чтобы указать выражению выбросить исключение при вычислении, используйте макрос `BOOST_REQUIRE_THROW`, который похож на макрос `REQUIRE_THROWS` в Catch, за исключением того, что нужно также указать тип исключения, которое необходимо создать. Его использование заключается в следующем:

```
BOOST_REQUIRE_THROW(выражение, желаемый-тип-исключения);
```

Если *выражение* не выдает исключение типа *желаемый-тип-исключения*, утверждение не выполнится.

Давайте рассмотрим, как выглядит пакетный тест AutoBrake с использованием Boost Test.

Рефакторинг теста initial_car_speed_is_zero в стиле Boost Test

Будем использовать намеренно сломанный `AutoBrake` в листинге 10.32 с отсутствующим инициализатором члена для `speed_mps`. Листинг 10.42 заставляет Boost Test работать с провальным юнит-тестом.

Листинг 10.42. Намеренное закомментирование инициализатора члена speed_mps для провала теста (с использованием Boost Test)

```
#define BOOST_TEST_MODULE AutoBrakeTest ❶
#include <boost/test/unit_test.hpp>
#include <functional>

struct IServiceBus {
  --пропуск--
};
```

```
struct MockServiceBus : IServiceBus {
  --пропуск--
};

struct AutoBrake {
  AutoBrake(IServiceBus& bus)
    : collision_threshold_s{ 5 }/*,
      speed_mps{} */❷ {
  --пропуск--
};

BOOST_AUTO_TEST_CASE(InitialCarSpeedIsZero❸) {
  MockServiceBus bus{};
  AutoBrake auto_brake{ bus };
  BOOST_TEST(0 == auto_brake.get_speed_mps()); ❹
}
-------------------------------------------------------------------------------
Running 1 test case...
C:/Users/josh/projects/cpp-book/manuscript/part_2/10-testing/samples/boost/
minimal.cpp(80): error: in "InitialCarSpeedIsZero": check 0 == auto_brake.
get_speed_mps() has failed [0 != -9.2559631349317831e+61] ❺
*** 1 failure is detected in the test module "AutoBrakeTest"
```

Имя тестового модуля – `AutoBrakeTest` ❶. После комментирования инициализатора члена `speed_mps` ❷ получим тест `InitialCarSpeedIsZero` ❸. Утверждение `BOOST_TEST` проверяет, является ли `speed_mps` равным нулю ❹. Как в случае с Catch и Google Test, вы получаете информативное сообщение об ошибке, которое говорит, что пошло не так ❺.

Тестовые приспособления

Как и Google Test, Boost Test работает с общим кодом установки, используя понятие тестовых приспособлений. Использовать их так же просто, как объявить объект RAII, в котором логика настройки для теста содержится в конструкторе этого класса, а логика разборки содержится в деструкторе. В отличие от Google Test, не нужно наследовать родительский класс в тестовом приспособлении. Тестовые приспособления работают с любой пользовательской структурой.

Для использования тестового приспособления в юнит-тесте используется макрос `BOOST_FIXTURE_TEST _CASE`, который принимает два параметра. Первый параметр — это имя юнит-теста, а второй параметр — класс тестового приспособления. Внутри тела макроса реализуется юнит-тест, как если бы это был метод класса тестовых приспособлений, как показано в листинге 10.43.

Листинг 10.43. Пример использования тестового приспособления Boost

```
#define BOOST_TEST_MODULE TestModuleName
#include <boost/test/unit_test.hpp>

struct MyTestFixture { }; ❶
```

```
BOOST_FIXTURE_TEST_CASE❷(MyTestA❸, MyTestFixture) {
  // Тест A
}

BOOST_FIXTURE_TEST_CASE(MyTestB❹, MyTestFixture) {
  // Тест B
}
-------------------------------------------------------------------------------
Running 2 test cases...

*** No errors detected
```

Здесь класс `MyTestFixture` определяется ❶ и используется в качестве второго параметра для каждого экземпляра `BOOST_FIXTURE_TEST_CASE` ❷. Объявляются два юнит-теста: `MyTestA` ❸ и `MyTestB` ❹. Любая настройка, которая выполняется в `MyTestFixture`, влияет на каждый `BOOST_FIXTURE_TEST_CASE`.

Далее будем использовать приспособления Boost Test для повторной реализации набора тестов `AutoBrake`.

Рефакторинг юнит-тестов AutoBrake с Boost Test

В листинге 10.44 реализован пакет юнит-тестов AutoBrake с использованием тестового приспособления Boost Test

Листинг 10.44. Использование Boost Test для реализации юнит-тестов

```
#define BOOST_TEST_MODULE AutoBrakeTest
#include <boost/test/unit_test.hpp>
#include <functional>

struct IServiceBus {
  --пропуск--
};

struct MockServiceBus : IServiceBus {
  --пропуск--
};

struct AutoBrakeTest { ❶
  MockServiceBus bus{};
  AutoBrake auto_brake{ bus };
};

BOOST_FIXTURE_TEST_CASE❷(InitialCarSpeedIsZero, AutoBrakeTest) {
  BOOST_TEST(0 == auto_brake.get_speed_mps());
}

BOOST_FIXTURE_TEST_CASE(InitialSensitivityIsFive, AutoBrakeTest) {
  BOOST_TEST(5 == auto_brake.get_collision_threshold_s());
}

BOOST_FIXTURE_TEST_CASE(SensitivityGreaterThanOne, AutoBrakeTest) {
```

```
  BOOST_REQUIRE_THROW(auto_brake.set_collision_threshold_s(0.5L),
                      std::exception);
}

BOOST_FIXTURE_TEST_CASE(SpeedIsSaved, AutoBrakeTest) {
  bus.speed_update_callback(SpeedUpdate{ 100L });
  BOOST_TEST(100 == auto_brake.get_speed_mps());
  bus.speed_update_callback(SpeedUpdate{ 50L });
  BOOST_TEST(50 == auto_brake.get_speed_mps());
  bus.speed_update_callback(SpeedUpdate{ 0L });
  BOOST_TEST(0 == auto_brake.get_speed_mps());
}

BOOST_FIXTURE_TEST_CASE(NoAlertWhenNotImminent, AutoBrakeTest) {
  auto_brake.set_collision_threshold_s(2L);
  bus.speed_update_callback(SpeedUpdate{ 100L });
  bus.car_detected_callback(CarDetected{ 1000L, 50L });
  BOOST_TEST(0 == bus.commands_published);
}

BOOST_FIXTURE_TEST_CASE(AlertWhenImminent, AutoBrakeTest) {
  auto_brake.set_collision_threshold_s(10L);
  bus.speed_update_callback(SpeedUpdate{ 100L });
  bus.car_detected_callback(CarDetected{ 100L, 0L });
  BOOST_TEST(1 == bus.commands_published);
  BOOST_TEST(1L == bus.last_command.time_to_collision_s);
}
-------------------------------------------------------------------------------
Running 6 test cases...

*** No errors detected
```

Определяется класс тестового приспособления `AutoBrakeTest` для настройки `AutoBrake` и `MockServiceBus` ❶. Он идентичен тестовому приспособлению Google Test, за исключением того что не нужно наследовать от родительских классов, выпущенных платформой. Каждый юнит-тест представляется с помощью макроса `BOOST_FIXTURE_TEST_CASE` ❷. Остальные тесты используют макросы утверждений `BOOST_TEST` и `BOOST_REQUIRE_THROW`; в противном случае тесты выглядят очень похожими на тесты Catch. Вместо `TEST_CASE` и элементов `SECTION` предоставляются классы тестовых приспособлений и `BOOST_FIXTURE_TEST_CASE`.

Итоги: фреймворки тестирования

Несмотря на то что в этом разделе были представлены три различные структуры юнит-тестирования, доступны десятки высококачественных вариантов. Ни один из них не является универсальным. Большинство фреймворков поддерживают один и тот же базовый набор функций, тогда как некоторые из более продвинутых функций будут иметь разнородную поддержку. Стоит выбирать фреймворк для юнит-тестирования, основанный на том стиле, который удобен вам.

Фреймворки имитации

Только что изученные фреймворки юнит-тестирования будут работать в широком диапазоне настроек. Например, было бы вполне возможно создать интеграционные тесты, приемочные тесты, юнит-тесты и даже тесты производительности, используя Google Test.

Среды тестирования поддерживают широкий спектр стилей программирования, и их создатели лишь скромно придерживаются собственного мнения, как стоит разрабатывать программное обеспечение, чтобы сделать его тестируемым.

Фреймворки имитации немного более своевольны, чем фреймворки юнит-тестирования. В зависимости от структуры имитации нужно следовать определенным рекомендациям по дизайну, чтобы определить, как классы зависят друг от друга. Класс `AutoBrake` использовал современный шаблон проектирования, называемый *внедрением зависимостей*. Класс `AutoBrake` зависит от `IServiceBus`, который был введен с помощью конструктора `AutoBrake`. Также `IServiceBus` был представлен как интерфейс. Существуют другие методы для достижения полиморфного поведения (например, шаблоны), и каждый включает компромиссы.

Все фреймворки имитации, рассматриваемые в этом разделе, очень хорошо работают с внедрением зависимостей. В той или иной степени фреймворки имитации устраняют необходимость определять свои собственные имитации. Напомним, что вы реализовали `MockServiceBus`, чтобы тестировать модуль `AutoBrake`, как показано в листинге 10.45.

Листинг 10.45. Собственноручно собранный MockServiceBus

```
struct MockServiceBus : IServiceBus {
  void publish(const BrakeCommand& cmd) override {
    commands_published++;
    last_command = cmd;
  };
  void subscribe(SpeedUpdateCallback callback) override {
    speed_update_callback = callback;
  };
  void subscribe(CarDetectedCallback callback) override {
    car_detected_callback = callback;
  };
  BrakeCommand last_command{};
  int commands_published{};
  SpeedUpdateCallback speed_update_callback{};
  CarDetectedCallback car_detected_callback{};
};
```

Каждый раз, когда нужно добавить юнит-тест, включающий какое-то новое взаимодействие с `IServiceBus`, вероятно, потребуется обновить класс `MockServiceBus`. Это утомительно и подвержено ошибкам. Кроме того, неясно, можно ли поделиться этим классом с другими командами: в нем реализована большая часть вашей собственной логики, которая не будет очень полезна, скажем, команде датчиков давления в шинах.

Кроме того, каждый тест может иметь разные требования. Фреймворки имитации позволяют определять имитационные классы, часто используя макросы или магию шаблонов. В каждом юнит-тесте можно настроить имитацию специально для этого теста. С одним определением имитации это было бы чрезвычайно сложно сделать.

Такое отделение объявления имитации от ее определения для конкретного теста эффективно по двум причинам. Во-первых, можно определить различные виды поведения для каждого юнит-теста. Это позволяет, например, моделировать исключительные условия для одних юнит-тестов, но не для других. Во-вторых, это делает юнит-тесты гораздо более конкретными. Размещение поведения пользовательской имитации в юнит-тесте, а не в отдельном исходном файле делает цель теста гораздо более понятной для разработчика.

Чистый эффект от использования фреймворка состоит в том, что он делает имитирование намного менее проблематичным. Когда имитация проста, то становится возможным хорошее юнит-тестирование (и TDD). Без имитации юнит-тестирование может быть очень сложным; тесты могут быть медленными, ненадежными и хрупкими из-за медленных или подверженных ошибкам зависимостей. Как правило, предпочтительнее, например, использовать имитацию соединения с базой данных вместо полноценного производственного экземпляра при попытках использовать TDD для реализации новых функций в классе.

Этот раздел содержит обзор двух фреймворков имитации, Google Mock и HippoMocks, и включает в себя краткое упоминание двух других, FakeIt иTrompeloeil. По техническим причинам, связанным с отсутствием генерации кода во время компиляции, создание фреймворка имитации гораздо сложнее в C++, чем в большинстве других языков, особенно в тех, которые имеют отражение типов — возможность языка, которая позволяет коду программно рассуждать об информации о типах. Следовательно, существует множество высококачественных фреймворков, каждый из которых имеет свои собственные компромиссы из-за фундаментальных трудностей, связанных с имитацией в C++.

Google Mock

Один из самых популярных фреймворков — Google C++ Mocking Framework (или Google Mock), который входит в состав Google Test. Это один из самых старых и многофункциональных фреймворков. Если Google Test уже установлен, включить Google Mock очень просто. Во-первых, убедитесь, что статическая библиотека gmock включена в редактор связей, как это было сделано для `gtest` и `gtest_main`. Затем добавьте `#include"gmock/gmock.h"`.

Если вы используете Google Test в качестве основы для юнит-тестирования, то делать больше ничего не нужно. Google Mock будет работать без проблем со своей родственной библиотекой. Если используется другая платформа юнит-тестирования, потребуется предоставить код инициализации в точке входа двоичного файла, как показано в листинге 10.46.

Листинг 10.46. Добавление Google Mock в фреймворк юнит-тестирования

```
#include "gmock/gmock.h"

int main(int argc, char** argv) {
  ::testing::GTEST_FLAG(throw_on_failure) = true; ❶
  ::testing::InitGoogleMock(&argc, argv); ❷
  // Обычный юнит-тест. Google Mock инициализирован
}
```

`GTEST_FLAG throw_on_failure` ❶ заставляет Google Mock генерировать исключение при сбое какого-либо утверждения, связанного с имитацией. Вызов `InitGoogleMock` ❷ использует аргументы командной строки для выполнения любых необходимых настроек (более подробную информацию см. в документации по Google Mock).

Имитация интерфейса

Для каждого интерфейса, который нужно смоделировать, существует какая-либо формальность. Нужно взять каждую функцию `virtual` интерфейса и преобразовать ее в макрос. Для не-`const` методов используется `MOCK_METHOD*`, а для `const` — `MOCK_CONST_METHOD*`, заменяя * на количество параметров, которые принимает функция. Первый параметр `MOCK_METHOD` — это имя виртуальной функции. Второй параметр — это прототип функции. Например, чтобы создать имитацию `IServiceBus`, нужно построить определение, показанное в листинге 10.47.

Листинг 10.47. Google Mock для MockServiceBus

```
struct MockServiceBus : IServiceBus { ❶
  MOCK_METHOD1❷(publish❸, void(const BrakeCommand& cmd)❹);
  MOCK_METHOD1(subscribe, void(SpeedUpdateCallback callback));
  MOCK_METHOD1(subscribe, void(CarDetectedCallback callback));
};
```

Начало определения `MockServiceBus` идентично определению любой другой реализации `IServiceBus` ❶. `MOCK_METHOD` реализуется три раза ❷. Первый параметр ❸ — это имя виртуальной функции, а второй параметр ❹ — это прототип функции.

Немного утомительно генерировать эти определения самостоятельно. В определении `MockServiceBus` нет дополнительной информации, которая еще не доступна в `IServiceBus`. Хорошо это или плохо, но это одна из издержек использования Google Mock. Можно отказаться от создания этого шаблона, используя инструмент `gmock_gen.py`, включенный в папку `scripts/generator` дистрибутива Google Mock. Для этого понадобится Python 2, и он не гарантированно работает во всех ситуациях. См. документацию Google Mock для получения дополнительной информации.

Теперь, после определения `MockServiceBus`, можно использовать его в юнит-тестах. В отличие от имитации, определенной самостоятельно, можно настроить Google Mock специально для каждого юнит-теста. В этой конфигурации предоставляется невероятная гибкость. Ключом к успешной имитации конфигурации является использование соответствующих ожиданий.

Ожидания

Ожидание похоже на утверждение для объекта имитации; в нем выражены обстоятельства, при которых имитация ожидает вызова, и то, что она должна делать в ответ. Обстоятельства указываются с использованием объектов, называемых *обнаружителями совпадений*. Часть «что она должен делать в ответ» называется *действием*. В следующих разделах будет представлена каждая из этих концепций.

Ожидания объявляются с помощью макроса `EXPECT_CALL`. Первый параметр этого макроса — объект имитации, а второй — ожидаемый вызов метода. Этот вызов метода может дополнительно содержать обнаружители совпадений для каждого параметра. Они помогают Google Mock определять, квалифицируется ли вызов конкретного метода как ожидаемый вызов. Формат их следующий:

```
EXPECT_CALL(объект_имитации, метод(обнаружители-совпадений))
```

Есть несколько способов сформулировать утверждения об ожиданиях, и ваш выбор зависит от того, насколько строги требования к взаимодействию тестируемого устройства с имитацией. Вас волнует, вызывает ли код неожидаемые функции имитации? Это действительно зависит от приложения. Вот почему есть три варианта: нудный, красивый и строгий.

Нудная имитация выбирается по умолчанию. Если вызывается функция Naggy Mock и нет `EXPECT_CALL`, соответствующего вызову, Google Mock выведет предупреждение о «неинтересном вызове», но тест не пройдет только из-за неинтересного вызова. Можно просто добавить `EXPECT_CALL` в тест как быстрое исправление, чтобы подавить предупреждение о неинтересном вызове, потому что тогда вызов перестает быть неожиданным.

В некоторых ситуациях может быть слишком много неинтересных вызовов. В таких случаях стоит использовать *красивую имитацию*. Красивая имитация не выдаст предупреждения о неинтересных вызовах.

Если вас беспокоит какое-либо взаимодействие с имитацией, которое вы не учли, можно использовать *строгую имитацию*. Строгие имитации не пройдут тест, если будет сделан какой-либо вызов имитации, для которого нет соответствующего `EXPECT_CALL`.

Каждый из этих типов имитации является шаблоном класса. Способ создания этих классов прост, как показано в листинге 10.48.

Листинг 10.48. Три разных стиля Google Mock

```
MockServiceBus naggy_mock;❶
::testing::NiceMock<MockServiceBus> nice_mock;❷
::testing::StrictMock<MockServiceBus> strict_mock;❸
```

Нудная имитация ❶ выбирается по умолчанию. Каждый `::testing::NiceMock` ❷ и `::testing::StrictMock` ❸ принимает один параметр шаблона, класс используемой имитации. Все три параметра являются совершенно действительными первыми параметрами для `EXPECT_CALL`.

Как правило, стоит использовать красивые имитации. Использование нудных и строгих имитаций может привести к очень хрупкому тестированию. При использовании строгой имитации подумайте, действительно ли необходимо быть настолько ограниченным в отношении взаимодействия тестируемого устройства с имитацией.

Второй параметр в `EXPECT_CALL` — это имя метода, вызов которого ожидается, за которым следуют параметры, которые используются в этом вызове. Иногда это легко. В других случаях есть более сложные условия, которые нужно выразить, при этом вызовы могут совпадать и не совпадать. В таких ситуациях используются обнаружители совпадений.

Обнаружители совпадений

Когда метод имитации принимает аргументы, у вас есть возможность проверить, соответствует ли вызов ожидаемому. В простых случаях можно использовать буквальные значения. Если метод имитации вызывается с точно указанным литеральным значением, вызов соответствует ожиданию; в противном случае это не так. С другой стороны, можно использовать объект Google `Mock::testing::_`, который сообщает Google Mock о любом совпадении значения.

Предположим, например, что нужно вызвать `publish` и вам все равно, какой аргумент использовать. `EXPECT_CALL` в листинге 10.49 имел бы соответствующий вид.

Листинг 10.49. Использование обнаружителя совпадений ::testing::_ в ожидании

```
--пропуск--
using ::testing::_; ❶

TEST(AutoBrakeTest, PublishIsCalled) {
  MockServiceBus bus;
  EXPECT_CALL(bus, publish(_❷));
  --пропуск--
}
```

Чтобы сделать юнит-тест красивым и точным, можно задействовать метод `for::testing::_` ❶. Символ _ используется, чтобы сообщить Google Mock, что любой вызов `publish` с одним аргументом будет соответствовать ❷.

Несколько более избирательным обнаружителем совпадений является шаблон класса `::testing::A`, который будет указывать на совпадение только в том случае, если метод вызывается с параметром определенного типа. Этот тип выражается как параметр шаблона для `A`, поэтому `A<MyType>` будет соответствовать только параметру типа `MyType`. В листинге 10.50 изменение листинга 10.49 показывает более ограниченное ожидание, которое требует использования `BrakeCommand` в качестве параметра для `publish`.

Опять же используется `using` ❶ и `A<BrakeCommand>`, чтобы указать, что только `BrakeCommand` будет соответствовать этому ожиданию.

Листинг 10.50. Использование обнаружителя совпадений ::test::A в ожидании

```
--пропуск--
using ::testing::A; ❶

TEST(AutoBrakeTest, PublishIsCalled) {
  MockServiceBus bus;
  EXPECT_CALL(bus, publish(A<BrakeCommand>()❷));
  --пропуск--
}
```

Еще один обнаружитель совпадений, `::testing::Field`, позволяет проверять поля на аргументы, передаваемые в имитацию. Обнаружитель совпадения `Field` принимает два параметра: указатель на поле, которое нужно ожидать, а затем другой обнаружитель совпадения, чтобы выразить, соответствует ли указанное поле критериям. Предположим, нужно быть более конкретным в отношении вызова для `publish` ❷: необходимо указать, что `time_to_collision_s` равно 1 секунде. Можно выполнить эту задачу с помощью рефакторинга листинга 10.49, как показано в листинге 10.51.

Листинг 10.51. Использование обнаружителя совпадений Field в ожидании

```
--пропуск--
using ::testing::Field; ❶
using ::testing::DoubleEq; ❷

TEST(AutoBrakeTest, PublishIsCalled) {
  MockServiceBus bus;
  EXPECT_CALL(bus, publish(Field(&BrakeCommand::time_to_collision_s❸,
                                 DoubleEq(1L)❹)));
  --пропуск--
}
```

`using` применяется для `Field` ❶ и `DoubleEq` ❷, чтобы немного очистить код ожидания. Обнаружитель совпадений `Field` берет указатель на поле, которое вас интересует в `time_to_collision_s` ❸, и другой обнаружитель совпадений, который решает, соответствует ли поле критериям `DoubleEq` ❹.

Доступно много других обнаружителей совпадений, и они приведены в табл. 10.2. Но обратитесь к документации Google Mock для изучения всех деталей об их использовании.

ПРИМЕЧАНИЕ

Одной из полезных особенностей обнаружителей совпадений является то, что можно использовать их в качестве альтернативного вида утверждения для юнит-тестов. Альтернативный макрос — это `EXPECT_THAT(значение, совпадение)` или `ASSERT_THAT(значение, обнаружитель совпадения)`. Например, можно заменить утверждение

```
ASSERT_GT(power_level, 9000);
```

на более синтаксически приятное

```
ASSERT_THAT(power_level, Gt(9000));
```

Таблица 10.2. Обнаружители совпадений в Google Mock

Обнаружитель совпадений	Указывает на совпадение, когда аргумент...
_	Любое значение подходящего типа
A<тип>()	Значение данного типа
An<тип>()	Значение данного типа
Ge(значение)	Больше или равен значению
Gt(значение)	Больше значения
Le(значение)	Меньше или равен значению
Lt(значение)	Меньше значения
Ne(значение)	Не равен значению
IsNull()	Null
NotNull()	Не null
Ref(переменная)	Ссылка на переменную
DoubleEq(переменная)	Значение double, которое приблизительно равно переменной
FloatEq(переменная)	Значение float, которое приблизительно равно переменной
EndsWith(стр)	Строка, заканчивающаяся на стр
HasSubstr(стр)	Строка, содержащая подстроку стр
StartsWith(стр)	Строка, начинающаяся со стр
StrCaseEq(стр)	Строка, равная стр (без учета регистра)
StrCaseNe(стр)	Строка, не равная стр (без учета регистра)
StrEq(стр)	Строка, равная стр
StrNeq(стр)	Строка, не равная стр

Можно использовать EXPECT_CALL с StrictMock, чтобы обеспечить взаимодействие тестируемого модуля с имитацией. Но также можно указать, сколько раз имитация должна отвечать на вызов. Это называется *кардинальностью* ожидания.

Кардинальность

Возможно, наиболее распространенным методом для определения количества элементов является Times, в котором указывается, сколько раз следует ожидать вызова имитации. Метод Times принимает один параметр, который может быть целочисленным литералом или одной из функций, перечисленных в табл. 10.3.

Листинг 10.52 развивает листинг 10.51, чтобы указать, что publish должен быть вызван только один раз.

Таблица 10.3. Список спецификаторов кардинальности в Google Mock

Кардинальность	Определяет, что метод будет вызван ...
`AnyNumber()`	Сколько угодно раз
`AtLeast(n)`	Минимум `n` раз
`AtMost(n)`	Максимум `n` раз
`Between(m, n)`	Между `m` и `n` раз
`Exactly(n)`	Ровно `n` раз

Листинг 10.52. Использование спецификатора кардинальности Times в ожидании

```
--пропуск--
using ::testing::Field;
using ::testing::DoubleEq;

TEST(AutoBrakeTest, PublishIsCalled) {
  MockServiceBus bus;
  EXPECT_CALL(bus, publish(Field(&BrakeCommand::time_to_collision_s,
                                 DoubleEq(1L)))).Times(1)❶;
  --пропуск--
}
```

Вызов `Times` ❶ гарантирует, что `publish` будет вызван ровно один раз (независимо от того, какая имитация используется: красивая, строгая или нудная).

ПРИМЕЧАНИЕ

Как вариант, можно указать `Times(Exactly(1))`.

Теперь при наличии некоторых инструментов для определения критериев и количества элементов для ожидаемого вызова можно настроить, как имитация должна отвечать ожиданиям. Для этого используются действия.

Действия

Как и в случае с кардинальностью, все действия связаны с операторами `EXPECT_CALL`. Эти операторы могут помочь уточнить, сколько раз имитация ожидает вызова, какие значения возвращать каждый раз, когда она вызывается, и любые побочные эффекты (например, создание исключения), которые она должна выполнять. Действия `WillOnce` и `WillRepeatedly` указывают, что имитация должна делать в ответ на запрос. Эти действия могут быть довольно сложными, но для краткости этот раздел охватывает два случая. Во-первых, можно использовать конструкцию `Return` для возврата значений вызывающей стороне:

```
EXPECT_CALL(jenny_mock, get_your_number()) ❶
  .WillOnce(Return(8675309)) ❷
  .WillRepeatedly(Return(911)); ❸
```

EXPECT_CALL устанавливается обычным способом, а затем помечаются некоторые действия, которые указывают, какое значение jenny_mock будет возвращать каждый раз при вызове get_your_number ❶. Они читаются последовательно слева направо, поэтому первое действие, WillOnce ❷, указывает, что при первом вызове get_your_number значение jenny_mock возвращает значение 8675309. Следующее действие, WillRepeatedly ❸, указывает, что для всех последующих вызовов будет возвращено значение 911.

Поскольку IServiceBus не возвращает никаких значений, нужно, чтобы действие было немного более сложным. Для настраиваемого поведения можно использовать конструкцию Invoke, которая позволяет передавать Invocable, который будет вызываться с точными аргументами, передаваемыми в метод имитации. Допустим, нужно сохранить ссылку на функцию обратного вызова, которую AutoBrake регистрирует посредством subscribe. Это можно легко сделать с помощью Invoke, как показано в листинге 10.53.

Листинг 10.53. Использование Invoke для сохранения ссылки на обратный вызов subscribe, зарегистрированный AutoBrake

```
CarDetectedCallback callback; ❶
EXPECT_CALL(bus, subscribe(A<CarDetectedCallback>()))
    .Times(1)
    .WillOnce(Invoke([&callback❷](const auto& callback_in❸) {
      callback = callback_in; ❹
    }));
```

Первый (и единственный) раз, когда subscribe вызывается с помощью CarDetectedCallback, действие WillOnce(Invoke(...)) будет вызывать лямбда-выражение, переданное в качестве параметра. Это лямбда-выражение захватывает CarDetectedCallback, объявленный ❶ по ссылке ❷. По определению лямбда-выражение имеет тот же прототип функции, что и функция subscribe, так что можно использовать auto-вывод типа ❸ для определения правильного типа для callback_in (это CarDetectedCallback). Наконец, callback_in присваивается callback ❹. Теперь можно передавать события тем, кто подписывается, просто вызывая callback ❶. Конструкция Invoke — швейцарский нож в действии, потому что можно выполнить произвольный код с полной информацией о параметрах вызова. *Параметры вызова* — это параметры, которые смоделированный метод получил во время выполнения.

Собираем все вместе

Пересматривая пакет тестирования AutoBrake, можно переопределить бинарный файл юнит-теста Google Test, чтобы использовать Google Mock, а не созданную вручную имитацию, как показано в листинге 10.54.

Здесь у вас фактически два разных тестовых приспособления: NiceAutoBrakeTest ❶ и StrictAutoBrakeTest ❷. Тест NiceAutoBrakeTest создает экземпляр NiceMock. Это полезно для InitialCarSpeedIsZero, InitialSensitivityIsFive и SensitivityGreaterThanOne, потому что не нужно проверять какие-либо значимые взаимодей-

ствия с имитацией; это не главная цель этих тестов. Но необходимо сосредоточиться на `AlertWhenImminent` и `NoAlertWhenNotImminent`. Каждый раз, когда событие публикуется или подписывается на тип, оно может иметь серьезные последствия в системе. Паранойя `StrictMock` здесь оправдана.

Листинг 10.54. Реализация юнит-тестов с помощью Google Mock, а не с созданной вручную имитацией

```
#include "gtest/gtest.h"
#include "gmock/gmock.h"
#include <functional>

using ::testing::_;
using ::testing::A;
using ::testing::Field;
using ::testing::DoubleEq;
using ::testing::NiceMock;
using ::testing::StrictMock;
using ::testing::Invoke;

struct NiceAutoBrakeTest : ::testing::Test { ❶
  NiceMock<MockServiceBus> bus;
  AutoBrake auto_brake{ bus };
};

struct StrictAutoBrakeTest : ::testing::Test { ❷
  StrictAutoBrakeTest() {
    EXPECT_CALL(bus, subscribe(A<CarDetectedCallback>())) ❸
      .Times(1)
      .WillOnce(Invoke([this](const auto& x) {
        car_detected_callback = x;
      }));
    EXPECT_CALL(bus, subscribe(A<SpeedUpdateCallback>())) ❹
      .Times(1)
      .WillOnce(Invoke([this](const auto& x) {
        speed_update_callback = x;
      }));;
  }
  CarDetectedCallback car_detected_callback;
  SpeedUpdateCallback speed_update_callback;
  StrictMock<MockServiceBus> bus;
};

TEST_F(NiceAutoBrakeTest, InitialCarSpeedIsZero) {
  ASSERT_DOUBLE_EQ(0, auto_brake.get_speed_mps());
}

TEST_F(NiceAutoBrakeTest, InitialSensitivityIsFive) {
  ASSERT_DOUBLE_EQ(5, auto_brake.get_collision_threshold_s());
}

TEST_F(NiceAutoBrakeTest, SensitivityGreaterThanOne) {
```

```
  ASSERT_ANY_THROW(auto_brake.set_collision_threshold_s(0.5L));
}

TEST_F(StrictAutoBrakeTest, NoAlertWhenNotImminent) {
  AutoBrake auto_brake{ bus };

  auto_brake.set_collision_threshold_s(2L);
  speed_update_callback(SpeedUpdate{ 100L });
  car_detected_callback(CarDetected{ 1000L, 50L });
}

TEST_F(StrictAutoBrakeTest, AlertWhenImminent) {
  EXPECT_CALL(bus, publish(
                       Field(&BrakeCommand::time_to_collision_s, DoubleEq{ 1L
}))
                   ).Times(1);
  AutoBrake auto_brake{ bus };

  auto_brake.set_collision_threshold_s(10L);
  speed_update_callback(SpeedUpdate{ 100L });
  car_detected_callback(CarDetected{ 100L, 0L });
}
```

В определении `StrictAutoBrakeTest` можно увидеть подход `WillOnce/Invoke` к сохранению обратных вызовов для каждой подписки ❸❹. Они используются в `AlertWhenImminent` и `NoAlertWhenNotImminent` для имитации событий, поступающих от служебной шины. Это дает ощущение красивых, чистых, лаконичных юнит-тестов, при том что под капотом происходит много логики имитации. Помните, что для проведения всего этого тестирования даже не требуется рабочая сервисная шина!

HippoMocks

Google Mock — это один из оригинальных C++ фреймворков имитации, и сегодня он по-прежнему популярен. HippoMocks — это альтернативный фреймворк, созданный Питером Бинделсом (Peter Bindels). Как библиотеку только для заголовков HippoMocks очень просто установить. Просто загрузите последнюю версию с GitHub (github.com/dascandy/hippomocks/). Нужно включить заголовок `"hippomocks.h"` в тесты. HippoMocks будет работать с любым фреймворком тестирования.

ПРИМЕЧАНИЕ

На момент выхода книги последняя версия HippoMocks — v5.0.

Чтобы создать макет с помощью HippoMocks, сначала создается экземпляр объекта `MockRespository`. По умолчанию все имитации, полученные из этого `MockRepository`, потребуют *строгого упорядочения* ожиданий. Строго упорядоченные ожидания приводят к сбою теста, если какое-то из ожиданий не вызывается в точном указанном порядке. Обычно это не то, что хотелось бы использовать. Чтобы изменить это по-

ведение по умолчанию, установите для поля `autoExpect` в `MockRepository` значение `false`:

```
MockRepository mocks;
mocks.autoExpect = false;
```

Теперь можно использовать `MockRepository` для создания макета `IServiceBus`. Это делается через функцию (член) шаблона `Mock`. Она вернет указатель на только что созданную имитацию:

```
auto* bus = mocks.Mock<IServiceBus>();
```

Здесь показано одно из главных выигрышных качеств `HippoMocks`: обратите внимание, что не нужно было генерировать шаблон на основе макросов для имитации `IServiceBus`, как это было сделано для Google Mock. Фреймворк может обрабатывать стандартные интерфейсы без каких-либо дополнительных усилий с вашей стороны.

Настроить ожидания также очень просто. Для этого используйте макрос `ExpectCall` в `MockRespository`. Он принимает два параметра: указатель на имитацию и указатель на ожидаемый метод:

```
mocks.ExpectCall(bus, IServiceBus::subscribe_to_speed)
```

Этот пример добавляет ожидание, что `bus.subscribe_to_speed` будет вызван. Имеются несколько обнаружителей совпадений, которые можно добавить к этому ожиданию, как показано в табл. 10.4.

Таблица 10.4. Обнаружители совпадений HippoMocks

Обнаружитель совпадений	Указывает на совпадение, когда ...
`With(аргументы)`	Параметры вызова соответствуют `аргументам`
`Match(предикат)`	`Предикат`, вызванный с параметрами вызова, возвращает `true`
`After(ожидание)`	`Ожидание` уже выполнилось (это полезно для обращений к уже произведенному вызову)

Можно определить действия, которые нужно выполнить в ответ на `ExpectCall`, как показано в табл. 10.5.

Таблица 10.5. Действия HippoMocks

Действие	Выполняет следующее после вызова
`Return(значение)`	Возвращает `значение` вызвавшему объекту
`Throw(исключение)`	Выбрасывает `исключение`
`Do(действие)`	Выполняет `действие` с параметрами вызова

По умолчанию HippoMocks требует, чтобы ожидание было выполнено ровно один раз (подобно кардинальности в Google Mock `.Times (1)`).

Например, можно выразить ожидание, что `publish` вызывается с помощью `BrakeCommand`, с заданием `time_to_collision_s` со значением 1.0, следующим образом:

```
mocks.ExpectCall❶(bus, IServiceBus::publish)
  .Match❷([](const BrakeCommand& cmd) {
    return cmd.time_to_collision_s == Approx(1); ❸
});
```

`ExpectCall` используется, чтобы указать, что шина должна вызываться с помощью метода `publish` ❶. Это ожидание уточняется с помощью обнаружителя совпадений `Match` ❷, который принимает предикат, принимающий те же аргументы, что и метод `publish`, — одну `const`-ссылку на `BrakeCommand`. Возвращается `true`, если поле `time_to_collision_s` в `BrakeCommand` равно 1.0; в противном случае возвращается `false` ❸, что полностью допустимо.

ПРИМЕЧАНИЕ

Начиная с версии 5.0, в HippoMocks нет встроенной поддержки для приближенных совпадений. Вместо этого использовался `Approx` из Catch ❸.

HippoMocks поддерживает перегрузки функций для свободных функций. Он также поддерживает перегрузки для методов, но их синтаксис не очень приятен для глаз. Если вы используете HippoMocks, лучше избегать перегрузок методов в интерфейсе, поэтому было бы лучше провести рефакторинг следующих строк в `IServiceBus`:

```
struct IServiceBus {
  virtual ~IServiceBus() = default;
  virtual void publish(const BrakeCommand&) = 0;
  virtual void subscribe_to_speed(SpeedUpdateCallback) = 0;
  virtual void subscribe_to_car_detected(CarDetectedCallback) = 0;
};
```

ПРИМЕЧАНИЕ

Некая философия проектирования утверждает, что нежелательно иметь перегруженный метод в интерфейсе, поэтому, если вы согласны с этим утверждением, отсутствие поддержки в HippoMocks будет спорным моментом.

Теперь `subscribe` больше не перегружен, и можно использовать HippoMocks. Листинг 10.55 изменяет набор тестов для использования HippoMocks с Catch.

Листинг 10.55. Перегрузка листинга 10.54 для использования HippoMocks и Catch вместо Google Mock и Google Test

```
#include "hippomocks.h"
--пропуск--
TEST_CASE("AutoBrake") {
  MockRepository mocks; ❶
```

```
  mocks.autoExpect = false;
  CarDetectedCallback car_detected_callback;
  SpeedUpdateCallback speed_update_callback;
  auto* bus = mocks.Mock<IServiceBus>();
  mocks.ExpectCall(bus, IServiceBus::subscribe_to_speed) ❷
    .Do([&](const auto& x) {
      speed_update_callback = x;
    });
  mocks.ExpectCall(bus, IServiceBus::subscribe_to_car_detected) ❸
    .Do([&](const auto& x) {
    car_detected_callback = x;
  });
  AutoBrake auto_brake{ *bus };

  SECTION("initializes speed to zero") {
    REQUIRE(auto_brake.get_speed_mps() == Approx(0));
  }

  SECTION("initializes sensitivity to five") {
    REQUIRE(auto_brake.get_collision_threshold_s() == Approx(5));
  }

  SECTION("throws when sensitivity less than one") {
    REQUIRE_THROWS(auto_brake.set_collision_threshold_s(0.5L));
  }


  SECTION("saves speed after update") {
    speed_update_callback(SpeedUpdate{ 100L }); ❹
    REQUIRE(100L == auto_brake.get_speed_mps());
    speed_update_callback(SpeedUpdate{ 50L });
    REQUIRE(50L == auto_brake.get_speed_mps());
    speed_update_callback(SpeedUpdate{ 0L });
    REQUIRE(0L == auto_brake.get_speed_mps());
  }

  SECTION("no alert when not imminent") {
    auto_brake.set_collision_threshold_s(2L);
    speed_update_callback(SpeedUpdate{ 100L }); ❺
    car_detected_callback(CarDetected{ 1000L, 50L });
  }

  SECTION("alert when imminent") {
    mocks.ExpectCall(bus, IServiceBus::publish) ❻
      .Match([](const auto& cmd) {
        return cmd.time_to_collision_s == Approx(1);
      });


    auto_brake.set_collision_threshold_s(10L);
    speed_update_callback(SpeedUpdate{ 100L });
    car_detected_callback(CarDetected{ 100L, 0L });
  }
}
```

ПРИМЕЧАНИЕ

Этот раздел связывает HippoMocks с Catch для демонстрационных целей, но HippoMocks работает со всеми фреймворками юнит-тестирования, описанными в этой главе.

Вы создаете `MockRepository` ❶ и ослабляете строгие требования к порядку, устанавливая `autoExpect` в `false`. После объявления двух обратных вызовов создается имитация `IServiceBus` (без определения класса имитации!), а затем устанавливаются ожидания ❷❸, которые будут соединять функции обратного вызова с `Autobrake`. Наконец, создается `auto_brake`, используя ссылку на имитацию шины.

Тесты `initializes speed to zero`, `initializes sensitivity to five` и `throws when sensitivity less thanone` не требуют дальнейшего взаимодействия с имитацией. На самом деле как строгая имитация `bus` не допустит дальнейшего взаимодействия без предупреждений. Поскольку HippoMocks не допускает красивых имитаций, как в Google Mock, это на самом деле принципиальная разница между листингом 10.54 и листингом 10.55.

В тесте `saves speed after update test` ❹ запускается серия обратных вызовов `speed_update` и утверждается, что скорости сохраняются правильно, как и раньше. Поскольку `bus` является строгой имитацией, также неявно утверждается, что с сервисной шиной дальнейшее взаимодействие не происходит.

В тесте `no alert when not imminent test` никаких изменений для `speed_update_callback` ❺ не требуется. Поскольку имитация строгая (и не ожидается публикации `BrakeCommand`), никаких дальнейших ожиданий тоже не требуется.

ПРИМЕЧАНИЕ

HippoMocks предлагает метод `NeverCall` в своих имитациях, который улучшит ясность тестов и ошибок при вызове.

Однако в тесте `alert when imminent` ожидается, что программа будет вызывать `publish` в `BrakeCommand`, поэтому это ожидание устанавливается ❻. Обнаружитель совпадений `Match` используется, чтобы обеспечить предикат, который проверяет, чтобы `time_to_collision_s` был приблизительно равен 1. Остальная часть теста такая же, как и раньше: в `AutoBrake` отправляется событие `SpeedUpdate` и последующее событие `CarDetected`, которое должно вызвать обнаружение столкновения.

HippoMocks — более упорядоченный фреймворк имитации, чем Google Mock. Он требует гораздо меньшего количества формальностей, но немного менее гибок.

ПРИМЕЧАНИЕ

Одна из областей, где HippoMocks является более гибким, чем Google Mock, — это функции без имитаций. HippoMocks может напрямую моделировать свободные функции и статические функции классов, тогда как Google Mock требует переписать код для использования интерфейса.

Несколько слов о других вариантах имитации: FakeIt и Trompeloeil

Доступен ряд других отличных фреймворков имитации. Но чтобы не затягивать и без того длинную главу, давайте кратко рассмотрим еще два фреймворка: FakeIt (от Эрана Пиера (Eran Pe'er), доступный по адресу github.com/eranpeer/FakeIt/) и Trompeloeil (от Бьорна Фаллера (Björn Fahller), доступный по адресу github.com/rollbear/trompeloeil/).

FakeIt похож на HippoMocks в своих кратких шаблонах использования, и это библиотека только для заголовков. Он отличается тем, что при построении ожиданий следует шаблону «запись по умолчанию». Вместо того чтобы указывать ожидания заранее, FakeIt проверяет, что методы имитации были правильно вызваны в *конце* теста. Действия, конечно, еще указаны в начале.

Хотя это абсолютно верный подход, я предпочитаю подход Google Mock/ HippoMocks для определения ожиданий и связанных с ними действий — все это в одном сжатом месте.

Trompeloeil (от французского *trompe-l'œil* — «обмануть глаз») можно считать современной заменой Google Mock. Как и в случае с Google Mock, для каждого из интерфейсов, которые можно сымитировать, требуется несколько шаблонов на основе макросов. В обмен на эти дополнительные усилия вы получаете множество мощных функций, в том числе установку тестовых переменных, возврат значений на основе параметров вызова и запрет определенных вызовов. Как Google Mock и HippoMocks, Trompeloeil требует заранее указать ожидания и действия (см. документацию для более подробной информации).

ПРИМЕЧАНИЕ

В оставшейся части книги примеры будут представлены с точки зрения юнит-тестов. Мне пришлось выбирать фреймворк для примеров, и я остановился на Catch по нескольким причинам. Во-первых, синтаксис Catch является наиболее лаконичным и его код проще всего привести в книге. В режиме только заголовка Catch компилируется намного быстрее, чем Boost Test. Это может считаться одобрением фреймворка (и это так), но я не намерен препятствовать использованию Google Test, Boost Test или любого другого фреймворка тестирования. Такие решения стоит принять после тщательного рассмотрения (и, надеюсь, некоторых экспериментов).

Итоги

В этой главе использовался расширенный пример построения системы автоматического торможения для автономного транспортного средства ради изучения основ TDD. Вы создали свою собственную среду тестирования и имитации, а затем узнали о многих преимуществах использования доступных фреймворков тестирования и имитации. Вы рассмотрели Catch, Google Test и Boost Test в качестве возможных фреймворков тестирования. Ради фреймворков имитации вы погрузились в Google

Mock и HippoMocks (с кратким упоминанием о FakeIt и Trompeloeil). Каждый из этих фреймворков имеет свои сильные и слабые стороны. Ваш выбор должен основываться главным образом на том, какие фреймворки способствуют вашей эффективности и продуктивности.

Упражнения

10.1. Ваша автомобильная компания завершила работу над сервисом, который определяет ограничение скорости на основе обозначений, наблюдаемых на обочине дороги. Команда по определению ограничения скорости будет периодически публиковать объекты следующего типа на шине событий:

```
struct SpeedLimitDetected {
  unsigned short speed_mps;
}
```

Служебная шина была расширена для включения этого нового типа:

```
#include <functional>
--пропуск--
using SpeedUpdateCallback = std::function<void(const SpeedUpdate&)>;
using CarDetectedCallback = std::function<void(const CarDetected&)>;
using SpeedLimitCallback = std::function<void(const SpeedLimitDetected&)>;

struct IServiceBus {
  virtual ~IServiceBus() = default;
  virtual void publish(const BrakeCommand&) = 0;
  virtual void subscribe(SpeedUpdateCallback) = 0;
  virtual void subscribe(CarDetectedCallback) = 0;
  virtual void subscribe(SpeedLimitCallback) = 0;
};
```

Обновите сервис новым интерфейсом и убедитесь, что тесты все еще проходят.

10.2. Добавьте приватное поле для последнего известного ограничения скорости. Реализуйте геттер для этого поля.

10.3. Владелец продукта хочет, чтобы вы установили последний известный предел скорости до 39 метров в секунду. Реализуйте юнит-тест, который проверяет вновь созданный `AutoBrake` с последним известным ограничением скорости (39).

10.4. Пройдите юнит-тесты.

10.5. Реализуйте юнит-тест, в котором публикуются три разных объекта `SpeedLimitDetected`, используя тот же метод обратного вызова, который был использован для `SpeedUpdate`, и `CarDetected`. После вызова каждого

из обратных вызовов проверьте последний известный предел скорости для объекта `AutoBrake`, чтобы убедиться, что значения совпадают.

10.6. Пройдите все юнит-тесты.

10.7. Проведите юнит-тест, в котором последний известный предел скорости составляет 35 метров в секунду, а скорость движения составляет 34 метра в секунду. Убедитесь, что `BrakeCommand` не публикуется через `AutoBrake`.

10.8. Пройдите все юнит-тесты.

10.9. Проведите юнит-тест, в котором последний известный предел скорости составляет 35 метров в секунду, а затем опубликуйте `SpeedUpdate` со скоростью 40 метров в секунду. Убедитесь, что выдается только одна команда `BrakeCommand`. Поле `time_to_collision_s` должно быть равно 0.

10.10. Пройдите все юнит-тесты.

10.11. Проведите новый юнит-тест, в котором последний известный предел скорости составляет 35 метров в секунду, а затем опубликуйте `SpeedUpdate` со скоростью 30 метров в секунду. Затем выполните `SpeedLimitDetected` со скоростью `speed_mps` 25 метров в секунду. Убедитесь, что выдается только одна команда `BrakeCommand`. Поле `time_to_collision_s` должно быть равно 0.

10.12. Пройдите все юнит-тесты.

Что еще почитать?

- «Specification by Example», Gojko Adzic (Manning, 2011)
- «BDD in Action», John Ferguson Smart (Manning, 2014)
- «Оптимизация программ на C++: проверенные методы повышения производительности», Курт Гантерот (Диалектика-Вильямс, 2016)
- «Принципы, паттерны и методики гибкой разработки на языке C#», Роберт Мартин (Символ-Плюс, 2011)
- «Экстремальное программирование. Разработка через тестирование», Кент Бек (Питер, 2017)
- «Growing Object-Oriented Software, Guided by Tests», Steve Freeman, Nat Pryce (Addison-Wesley, 2009)
- «Editor war». en.wikipedia.org/wiki/Editor_war
- «Tabs versus Spaces: An Eternal Holy War», Jamie Zawinski. www.jwz.org/doc/tabs-vs-spaces.html
- «Is TDD dead?», Martin Fowler. martinfowler.com/articles/ s-tdd-dead/

11 Умные указатели

Если вы хотите выполнить качественно несколько небольших задач, сделайте их сами. Если вы хотите совершать великие дела и оказывать большое влияние, учитесь делегировать.

Джон Максвелл

В этой главе вы изучите stdlib и библиотеки Boost. Эти библиотеки содержат набор умных указателей, которые управляют динамическими объектами с помощью парадигмы RAII, которую мы рассмотрели в главе 4. Они также способствуют созданию наиболее мощной модели управления ресурсами в любом языке программирования. Поскольку некоторые умные указатели используют *распределители* для настройки динамического выделения памяти, в этой главе также описывается, как предоставить пользовательский распределитель.

Умные указатели

Динамические объекты имеют наиболее гибкие времена жизни. С большой гибкостью приходит большая ответственность, поэтому нужно убедиться, что каждый динамический объект разрушается *ровно* один раз. Это может показаться не слишком сложным для небольших программ, но внешность обманчива. Просто подумайте, как исключения влияют на динамическое управление памятью. Каждый раз, когда может произойти ошибка или исключение, необходимо отслеживать, какие выделения были сделаны успешно, и обязательно высвобождать их в правильном порядке.

К счастью, для обработки такого скучного действия можно использовать RAII. Приобретая динамическое хранилище в конструкторе объекта RAII и освобождая динамическое хранилище в деструкторе, относительно трудно утратить (или дважды освободить) динамическую память. Это позволяет управлять временем жизни динамического объекта, используя семантику переноса и копирования.

Можно написать эти объекты RAII самостоятельно, но также можно использовать некоторые превосходные, заранее написанные реализации, называемые *умными указателями*. Умные указатели — это шаблоны классов, которые ведут себя как указатели и реализуют RAII для динамических объектов.

В этом разделе рассматриваются пять доступных опций, включенных в stdlib и Boost: ограниченные, уникальные, общие, слабые и навязчивые указатели. Их модели собственности различают эти пять категорий умных указателей.

Владение умными указателями

Каждый умный указатель имеет модель *владения*, которая определяет его связь с динамически размещаемым объектом. Когда умный указатель владеет объектом, срок его службы гарантированно будет по крайней мере таким же, как и у объекта. Иными словами, при использовании умных указателей вы можете быть уверены, что указанный объект жив и что не произойдет утечки памяти. Умный указатель управляет объектом, которым он владеет, поэтому нельзя забыть уничтожить его благодаря RAII.

Когда вы решаете, какой умный указатель использовать, требования к владельцу определяют выбор.

Ограниченные указатели

Указатель с областью действия выражает *непередаваемое исключительное* владение одним динамическим объектом. «Непередаваемое» означает, что указатели с областью действия не могут быть перемещены из одной области видимости в другую. Исключительное владение означает, что их нельзя скопировать, поэтому никакие другие умные указатели не могут владеть динамическим объектом указателя с областью действия. (Вспомните из раздела «Управление памятью», что область видимости — это то, где указатель виден программе.)

`boost::scoped_ptr` определяется в заголовке `<boost/smart_ptr/scoped_ptr.hpp>`.

ПРИМЕЧАНИЕ

Не существует указателя, ограниченного stdlib.

Создание

boost::scoped_ptr принимает один параметр шаблона, соответствующий указанному типу, как в boost::scoped_ptr<int> для типа «указатель на область видимости».

Все умные указатели, включая указатели ограниченные, имеют два режима: *пустой* и *полный*. Пустой умный указатель не имеет объекта и примерно аналогичен nullptr. Когда умный указатель создается по умолчанию, он пуст.

Ограниченный указатель предоставляет конструктор, принимающий необработанный указатель. (Указанный тип должен соответствовать параметру шаблона.) Это создает указатель с полной областью. Обычная идиома — создать динамический объект с помощью new и передать результат в конструктор, например так:

```
boost::scoped_ptr<PointedToType> my_ptr{ new PointedToType };
```

Эта строка динамически выделяет PointedToType и передает его указатель в конструктор ограниченного указателя.

Добавление OathBreakers

Чтобы изучить указатели области действия, давайте создадим пакет юнит-тестов Catch и класс DeadMenOfDunharrow, который отслеживает количество живых объектов, как показано в листинге 11.1.

Листинг 11.1. Настройка пакета юнит-тестов Catch с классом DeadMenOfDunharrow для исследования ограниченных указателей

```
#define CATCH_CONFIG_MAIN ❶
#include "catch.hpp" ❷
#include <boost/smart_ptr/scoped_ptr.hpp> ❸

struct DeadMenOfDunharrow { ❹
  DeadMenOfDunharrow(const char* m="") ❺
    : message{ m } {
    oaths_to_fulfill++; ❻
  }
  ~DeadMenOfDunharrow() {
    oaths_to_fulfill--; ❼
  }
  const char* message;
  static int oaths_to_fulfill;
};
int DeadMenOfDunharrow::oaths_to_fulfill{};
using ScopedOathbreakers = boost::scoped_ptr<DeadMenOfDunharrow>; ❽
```

Сначала объявляется CATCH_CONFIG_MAIN, чтобы Catch предоставил точку входа ❶ и включил заголовок Catch ❷, а затем заголовок Boost ограниченного указателя ❸. Затем объявляется класс DeadMenOfDunharrow ❹, который принимает необязательную строку с нулевым символом в конце, которая сохраняется в поле message ❺. Поле staticint с именем oaths_to_fulfill отслеживает, сколько объ-

ектов `DeadMenOfDunharrow` было построено. Соответственно, значение увеличивается в конструкторе ❻ и уменьшается в деструкторе ❼. Наконец, объявляется псевдоним типа `ScopedOathbreakers` для удобства ❽.

> **ЛИСТИНГИ CATCH**
>
> Теперь в большинстве листингов будут использоваться юнит-тесты Catch. Для краткости в листингах пропущена следующая формальность:
>
> ```
> #define CATCH_CONFIG_MAIN
> #include "catch.hpp"
> ```
>
> Все листинги, содержащие `TEST_CASE`, требуют этой преамбулы.
>
> Кроме того, каждый тестовый случай в каждом листинге проходит, если в комментарии не указано иное. Опять же для краткости в листингах опускаются все выводы `All tests pass`.
>
> Наконец, тесты, в которых используются пользовательские типы, функции и переменные из предыдущего листинга, для краткости опустим.

Явное приведение логических типов на основе владения

Иногда нужно определить, владеет ли ограниченный указатель объектом или он пустой. Удобно, что `scoped_ptr` неявно приводится к `bool` в зависимости от статуса владения: `true`, если он владеет объектом, `false` в противном случае.

Листинг 11.2 показывает, как работает это неявное поведение приведения.

Листинг 11.2. boost::scoped_ptr неявно приводится к bool.

```
TEST_CASE("ScopedPtr evaluates to") {
  SECTION("true when full") {
    ScopedOathbreakers aragorn{ new DeadMenOfDunharrow{} }; ❶
    REQUIRE(aragorn); ❷
  }
  SECTION("false when empty") {
    ScopedOathbreakers aragorn; ❸
    REQUIRE_FALSE(aragorn); ❹
  }
}
```

При использовании конструктора, принимающего указатель ❶, `scoped_ptr` конвертируется в `true` ❷. При использовании конструктора по умолчанию ❸, `scoped_ptr` конвертируется в `false` ❹.

Обертка RAII

Когда `scoped_ptr` владеет динамическим объектом, он обеспечивает правильное управление им. В деструкторе `scoped_ptr` проверяется, владеет ли тот объектом. Если это так, деструктор `scoped_ptr` удаляет динамический объект.

Листинг 11.3 показывает это поведение, исследуя статическую переменную `oaths_to_fulfill` между инициализациями ограниченного указателя.

Листинг 11.3. boost::scoped_ptr — это обертка RAII

```
TEST_CASE("ScopedPtr is an RAII wrapper.") {
  REQUIRE(DeadMenOfDunharrow::oaths_to_fulfill == 0); ❶
  ScopedOathbreakers aragorn{ new DeadMenOfDunharrow{} }; ❷
  REQUIRE(DeadMenOfDunharrow::oaths_to_fulfill == 1); ❸
  {
    ScopedOathbreakers legolas{ new DeadMenOfDunharrow{} }; ❹
    REQUIRE(DeadMenOfDunharrow::oaths_to_fulfill == 2); ❺
  } ❻
  REQUIRE(DeadMenOfDunharrow::oaths_to_fulfill == 1); { ❼
}
```

В начале теста `oaths_to_fulfill` равен 0, потому что еще не был создан `DeadMenOfDunharrow` ❶. Вы создаете ограниченный указатель `aragorn` и передаете указатель на динамический объект `DeadMenOfDunharrow` ❷. Это увеличивает `oaths_to_fulfill` до 1 ❸. В рамках вложенной области объявляется еще один указатель с областью видимости — `legolas` ❹. Поскольку `aragorn` все еще жив, `oaths_to_fulfill` теперь равен 2 ❺. Когда внутренняя область видимости закрывается, `legolas` выпадает из области видимости и разрушается, забирая с собой `DeadMenOfDunharrow` ❻. Это уменьшает `DeadMenOfDunharrow` до 1 ❼.

Семантика указателей

Для удобства `scoped_ptr` реализует оператор разыменования * и оператор разыменования члена `->`, которые просто делегируют вызовы собственному динамическому объекту. Можно даже извлечь обычный указатель из `scoped_ptr` с помощью метода `get`, как показано в листинге 11.4.

Листинг 11.4. boost::scoped_ptr поддерживает семантику указателей

```
TEST_CASE("ScopedPtr supports pointer semantics, like") {
  auto message = "The way is shut";
  ScopedOathbreakers aragorn{ new DeadMenOfDunharrow{ message } }; ❶
  SECTION("operator*") {
    REQUIRE((*aragorn).message == message); ❷
  }
  SECTION("operator->") {
    REQUIRE(aragorn->message == message); ❸
  }
  SECTION("get(), which returns a raw pointer") {
    REQUIRE(aragorn.get() != nullptr); ❹
  }
}
```

`aragorn` создается с ограниченным указателем с сообщением `The way is shut` ❶, которое используется в трех отдельных сценариях для проверки семантики указателя. Во-первых, можно использовать оператор * для разыменования базового динами-

ческого объекта, на который нацелен указатель. В этом примере `aragorn` разыменовывается и извлекается сообщение, чтобы убедиться, что все так и работает ❷. Также можно использовать оператор `->` для разыменования членов ❸. Наконец, если нужен обычный указатель на динамический объект, можно использовать метод `get` для его извлечения ❹.

Сравнение с nullptr

Шаблон класса `scoped_ptr` реализует операторы сравнения `operator==` и `operator!=`, которые определяются только при сравнении `scoped_ptr` с нулевым значением. Функционально это практически идентично неявному преобразованию `bool`, как показано в листинге 11.5.

Листинг 11.5. boost::scoped_ptr поддерживает сравнение с nullptr

```
TEST_CASE("ScopedPtr supports comparison with nullptr") {
  SECTION("operator==") {
    ScopedOathbreakers legolas{};
    REQUIRE(legolas == nullptr); ❶
  }
  SECTION("operator!=") {
    ScopedOathbreakers aragorn{ new DeadMenOfDunharrow{} };
    REQUIRE(aragorn != nullptr); ❷
  }
}
```

Пустой ограниченный указатель равен (`==`) `nullptr` ❶, тогда как полный ограниченный указатель не равен (`!=`) `nullptr` ❷.

Обмен

Иногда нужно переключить динамический объект, принадлежащий `scoped_ptr`, на динамический объект, принадлежащий другому `scoped_ptr`. Это называется объектным обменом, и `scoped_ptr` содержит метод `swap`, который реализует это поведение, как показано в листинге 11.6.

Листинг 11.6. boost::scoped_ptr поддерживает swap

```
TEST_CASE("ScopedPtr supports swap") {
  auto message1 = "The way is shut.";
  auto message2 = "Until the time comes.";
  ScopedOathbreakers aragorn {
    new DeadMenOfDunharrow{ message1 } ❶
  };
  ScopedOathbreakers legolas {
    new DeadMenOfDunharrow{ message2 } ❷
  };
  aragorn.swap(legolas); ❸
  REQUIRE(legolas->message == message1); ❹
  REQUIRE(aragorn->message == message2); ❺
}
```

Создаются два объекта scoped_ptr – aragorn ❶ и legolas ❷, каждый из которых имеет свое сообщение. После вызова swap с aragorn и legolas ❸ они обмениваются динамическими объектами. Когда вы извлекаете их сообщения после обмена, то обнаруживаете, что они переключились ❹❺.

Сброс и замена scoped_ptr

Потребность уничтожения объекта, принадлежащего scoped_ptr, до уничтожения scoped_ptr возникает редко. Например, можно заменить принадлежащий ему объект новым динамическим объектом. С обеими этими задачами можно справиться с помощью перегруженного метода reset в scoped_ptr.

Если не предоставить аргумент, reset просто уничтожает принадлежащий объект.

Если вместо этого предоставить новый динамический объект в качестве параметра, reset сначала уничтожит принадлежащий в данный момент объект, а затем получит право собственности на параметр. Листинг 11.7 показывает такое поведение с одним тестом для каждого сценария.

Листинг 11.7. boost::scoped_ptr поддерживает reset

```
TEST_CASE("ScopedPtr reset") {
  ScopedOathbreakers aragorn{ new DeadMenOfDunharrow{} }; ❶
  SECTION("destructs owned object.") {
    aragorn.reset(); ❷
    REQUIRE(DeadMenOfDunharrow::oaths_to_fulfill == 0); ❸
  }
  SECTION("can replace an owned object.") {
    auto message = "It was made by those who are Dead.";
    auto new_dead_men = new DeadMenOfDunharrow{ message }; ❹
    REQUIRE(DeadMenOfDunharrow::oaths_to_fulfill == 2); ❺
    aragorn.reset(new_dead_men); ❻
    REQUIRE(DeadMenOfDunharrow::oaths_to_fulfill == 1); ❼
    REQUIRE(aragorn->message == new_dead_men->message); ❽
    REQUIRE(aragorn.get() == new_dead_men); ❾
  }
}
```

Первым шагом в обоих тестах является создание ограниченного aragorn-указателя, владеющего DeadMenOfDunharrow ❶. В первом тесте вызывается reset без аргумента ❷. Это приводит к тому, что ограниченный указатель разрушает свой собственный объект, и значение oaths_to_fulfill уменьшается до 0 ❸.

Во втором тесте создается новое динамически назначаемое new_dead_men с пользовательским message ❹. Это увеличивает oaths_to_fill до 2, потому что aragorn также все еще жив ❺. Затем вызывается reset с new_dead_men в качестве аргумента ❻, что делает две вещи:

- приводит к уничтожению оригинального DeadMenOfDunharrow, принадлежащего aragorn, что уменьшает значение oaths_to_fulfill до 1 ❼;
- new_dead_men используется как динамически распределяемый объект, принадлежащий aragorn. При разыменовывании поля message обратите внимание, что

оно совпадает с `message`, хранящимся в `new_dead_men` ❽. (Эквивалентно `aragorn.get()` возвращает `new_dead_men` ❾.)

Непереносимость

Нельзя перемещать или копировать `scoped_ptr`, что делает его непереносимым. Листинг 11.8 показывает, как попытка переместить или скопировать `scoped_ptr` приводит к некорректной работе программы.

Листинг 11.8. boost::scoped_ptr — непередаваемый. (Этот код не компилируется.)

```
void by_ref(const ScopedOathbreakers&) { } ❶
void by_val(ScopedOathbreakers) { } ❷

TEST_CASE("ScopedPtr can") {
  ScopedOathbreakers aragorn{ new DeadMenOfDunharrow };
  SECTION("be passed by reference") {
    by_ref(aragorn); ❸
  }
  SECTION("not be copied") {
    // НЕ КОМПИЛИРУЕТСЯ:
    by_val(aragorn); ❹
    auto son_of_arathorn = aragorn; ❺
  }
  SECTION("not be moved") {
    // НЕ КОМПИЛИРУЕТСЯ:
    by_val(std::move(aragorn)); ❻
    auto son_of_arathorn = std::move(aragorn); ❼
  }
}
```

Сначала объявляются фиктивные функции, которые принимают `scoped_ptr` по ссылке ❶ и по значению ❷. Вы все равно можете передать `scoped_ptr` по ссылке ❸, но попытка передать то же по значению не скомпилируется ❹. Кроме того, и попытку использовать конструктор копирования `scoped_ptr` или оператор присваивания копии ❺ не удастся скомпилировать. Кроме того, если вы попытаетесь переместить `scoped_ptr` с помощью `std::move`, код не скомпилируется ❻❼.

ПРИМЕЧАНИЕ

Как правило, использование `boost::scoped_ptr` не требует дополнительных затрат по сравнению с использованием обычного указателя.

boost::scoped_array

`boost::scoped_array` — это ограниченный указатель для динамических массивов. Он поддерживает то же использование, что и `boost::scoped_ptr`, но также реализует `operator[]`, поэтому можно взаимодействовать с элементами ограниченного массива так же, как и с обычным массивом. В листинге 11.9 приведена эта дополнительная функция.

Листинг 11.9. В boost::scoped_array реализован operator[]

```
TEST_CASE("ScopedArray supports operator[]") {
  boost::scoped_array<int❶> squares{
    new int❷[5] { 0, 4, 9, 16, 25 }
  };
  squares[0] = 1; ❸
  REQUIRE(squares[0] == 1); ❹
  REQUIRE(squares[1] == 4);
  REQUIRE(squares[2] == 9);
}
```

`scoped_array` объявляется так же, как `scoped_ptr`, используя единственный параметр шаблона ❶. В случае `scoped_array` параметр шаблона — это тип, содержащийся в массиве ❷, а не тип массива. Динамический массив передается конструктору `squares`, делая динамический массив `squares` владельцем массива. Можно использовать `operator[]` для записи ❸ и чтения элементов ❹.

Неполный список поддерживаемых операций

К этому моменту вы узнали об основных особенностях указателей в области видимости. В таблице 11.1 перечислены все обсуждаемые операторы, а также некоторые другие, которые еще не были рассмотрены. В таблице `ptr` — это простой указатель, а `s_ptr` — ограниченный. См. документацию Boost для получения дополнительной информации.

Таблица 11.1. Все поддерживаемые операции `boost::scoped_ptr`

Операция	Примечания
`scoped_ptr{ }` или `scoped_ptr { nullptr }`	Создает пустой ограниченный указатель
`scoped_ptr { ptr }`	Создает ограниченный указатель, владеющий динамическим объектом, на который указывает `ptr`
`~scoped_ptr()`	Вызывает `delete` на принадлежащем объекте, если является полным
`s_ptr1.swap(s_ptr2)`	Обмен принадлежащими объектами между `s_ptr1` и `s_ptr2`
`swap(s_ptr1, s_ptr2)`	Свободная функция, идентичная методу `swap`
`s_ptr.reset()`	Если полный, вызывает `delete` для объекта, принадлежащего `s_ptr`
`s_ptr.reset(ptr)`	Удаляет принадлежащий в настоящее время объект, а затем становится владельцем `ptr`
`ptr = s_ptr.get()`	Возвращает обычный указатель `ptr`; `s_ptr` сохраняет право владения
`*s_ptr`	Оператор разыменования на принадлежащем объекте
`s_ptr->`	Оператор разыменования члена на принадлежащем объекте
`bool{ s_ptr }`	Преобразование `bool`: `true`, если полный, `false`, если пустой

Уникальные указатели

Уникальный указатель имеет передаваемое исключительное право собственности на один динамический объект. Можно перемещать уникальные указатели, что делает их переносимыми. Они также имеют исключительную собственность, поэтому их нельзя скопировать. У stdlib есть `unique_ptr`, доступный в заголовке `<memory>`.

ПРИМЕЧАНИЕ

Boost не предоставляет уникальный указатель.

Создание

Функция `std::unique_ptr` принимает один параметр шаблона, соответствующий указанному типу, как в `std::unique_ptr<int>` для типа «уникальный указатель на `int`».

Как и в случае с ограниченным указателем, уникальный указатель имеет конструктор по умолчанию, который инициализирует уникальный указатель как пустой. Он также предоставляет конструктор, принимающий обычный указатель, который становится владельцем динамического объекта, на который указывает указатель. Один способ создания — создать динамический объект через `new` и передать результат в конструктор, например так:

```
std::unique_ptr<int> my_ptr{ new int{ 808 } };
```

Другой метод — использовать функцию `std::make_unique`. Функция `make_unique` — это шаблон, который принимает все аргументы и передает их соответствующему параметру конструктора шаблона. Это избавляет от необходимости использования `new`. Используя `std::make_unique`, можно переписать предыдущую инициализацию объекта следующим образом:

```
auto my_ptr = make_unique<int>(808);
```

Функция `make_unique` была создана, чтобы избежать некоторых утечек памяти, которые имели место, когда использовался `new` с предыдущими версиями C++. Однако в последней версии C++ эти утечки памяти больше не происходят. Использование конструктора в основном зависит от ваших предпочтений.

Поддерживаемые операции

Функция `std::unique_ptr` поддерживает все операции, которые поддерживает `boost::scoped_ptr`. Например, можно использовать следующий псевдоним типа в качестве замены для `ScopedOathbreaker` в листингах 11.1–11.7:

```
using UniqueOathbreakers = std::unique_ptr<DeadMenOfDunharrow>;
```

Одно из основных отличий между уникальными и ограниченными указателями состоит в том, что можно перемещать уникальные указатели, потому что они *переносимы*.

Переносимое и исключительное владение

Уникальные указатели не только могут быть переданы, но они имеют исключительное право собственности (их *нельзя* копировать). В листинге 11.10 показано, как можно использовать семантику переноса для `unique_ptr`.

Листинг 11.10. std::unique_ptr поддерживает семантику переноса для передачи владения

```
TEST_CASE("UniquePtr can be used in move") {
  auto aragorn = std::make_unique<DeadMenOfDunharrow>(); ❶
  SECTION("construction") {
    auto son_of_arathorn{ std::move(aragorn) }; ❷
    REQUIRE(DeadMenOfDunharrow::oaths_to_fulfill == 1); ❸
  }
  SECTION("assignment") {
    auto son_of_arathorn = std::make_unique<DeadMenOfDunharrow>(); ❹
    REQUIRE(DeadMenOfDunharrow::oaths_to_fulfill == 2); ❺
    son_of_arathorn = std::move(aragorn); ❻
    REQUIRE(DeadMenOfDunharrow::oaths_to_fulfill == 1); ❼
  }
}
```

Этот листинг создает `unique_ptr` с именем `aragorn` ❶, который используется в двух отдельных тестах.

В первом тесте `aragorn` перемещается с помощью `std::move` в конструктор переноса `son_of_arathorn` ❷. Поскольку `aragorn` передает право собственности на его `DeadMenOfDunharrow` в `son_of_arathorn`, объект `oaths_to_fulfill` все еще имеет значение 1 ❸.

Второй тест создает `son_of_arathorn` через `make_unique` ❹, который увеличивает значение `oaths_to_fulfill` до 2 ❺. Затем используется оператор присваивания переноса для перемещения `aragorn` в `son_of_arathorn` ❻. Опять же `aragorn` передает право собственности в `son_of_arathorn`. Поскольку `son_of_arathorn` может одновременно владеть только одним динамическим объектом, оператор присваивания переноса уничтожает принадлежащий в настоящее время объект перед очисткой динамического объекта `aragorn`. Это приводит к уменьшению `oaths_to_fulfill` до 1 ❼.

Уникальные массивы

В отличие от `boost::scoped_ptr`, `std::unique_ptr` имеет встроенную поддержку динамических массивов. Просто используется тип массива в качестве параметра шаблона в типе уникального указателя, как в `std::unique_ptr<int []>`.

Очень важно, чтобы `std::unique_ptr<T>` не был инициализирован динамическим массивом `T[]`. Это приведет к неопределенному поведению, потому что будет вызываться `delete` массива (а не `delete[]`). Компилятор не может уберечь от этого, потому что оператор `new[]` возвращает указатель, который неотличим от вида, возвращаемого оператором `new`.

Как и `scoped_array`, тип `unique_ptr` для массива предлагает `operator[]` для доступа к элементам. Листинг 11.11 демонстрирует эту концепцию.

Листинг 11.11. std::unique_ptr для типа массива поддерживает operator[]

```
TEST_CASE("UniquePtr to array supports operator[]") {
  std::unique_ptr<int[]❶> squares{
    new int[5]{ 1, 4, 9, 16, 25 } ❷
  };
  squares[0] = 1; ❸
  REQUIRE(squares[0] == 1); ❹
  REQUIRE(squares[1] == 4);
  REQUIRE(squares[2] == 9);
}
```

Параметр шаблона `int[]` ❶ указывает `std::unique_ptr`, что ему принадлежит динамический массив. Вы передаете вновь созданный динамический массив ❷, а затем используете `operator[]` для установки первого элемента ❸; затем используется `operator[]` для извлечения элементов ❹.

Удалители

У `std::unique_ptr` есть второй, необязательный, параметр шаблона, называемый его типом удалителя. *Удалитель* уникального указателя — это то, что вызывается, когда уникальный указатель должен уничтожить свой собственный объект.

Экземпляр `unique_ptr` содержит следующие параметры шаблона:

```
std::unique_ptr<T, Deleter=std::default_delete<T>>
```

Двумя параметрами шаблона являются `T`, тип принадлежащего динамического объекта, и `Deleter`, тип объекта, ответственного за освобождение принадлежащего объекта. По умолчанию `Deleter` — это `std::default_delete<T>`, который вызывает `delete` или `delete[]` для динамического объекта.

Чтобы написать пользовательский удалитель, все, что нужно, — это функциональный объект, который вызывается с помощью `T*`. (Уникальный указатель будет игнорировать возвращаемое значение удалителя.) Этот удалитель передается как второй параметр конструктору уникального указателя, что показано в листинге 11.12.

Тип принадлежащего объекта — `int` ❷, поэтому объявляется объект функции `my_deleter`, который принимает `int*` ❶. `decltype` используется, чтобы установить параметр шаблона удалителя ❸.

Листинг 11.12. Передача пользовательского удалителя в уникальный указатель

```
#include <cstdio>

auto my_deleter = [](int* x) { ❶
  printf("Deleting an int at %p.", x);
  delete x;
};
std::unique_ptr<int❷, decltype(my_deleter)❸> my_up{
  new int,
  my_deleter
};
```

Пользовательские удалители и системное программирование

Пользовательский удалитель используется, когда `delete` не обеспечивает требуемого поведения для освобождения ресурса. В некоторых разработках никогда не понадобится пользовательский удалитель. В других, таких как системное программирование, они могут оказаться весьма полезными. Рассмотрим простой пример, где происходит управление файлом с помощью низкоуровневых API-интерфейсов `fopen`, `fprintf` и `fclose` в заголовке `<cstdio>`.

Функция `fopen` открывает файл и имеет следующую сигнатуру:

```
FILE*❶ fopen(const char *имя-файла❷, const char *режим❸);
```

В случае успеха `fopen` возвращает ненулевой `FILE*` ❶. В случае неудачи `fopen` возвращает `nullptr` и устанавливает статическую переменную `int errno` равной коду ошибки, например отказ в доступе (`EACCES = 13`) или отсутствие такого файла (`ENOENT = 2`).

ПРИМЕЧАНИЕ

В заголовке `errno.h` приведен список всех состояний ошибок и их соответствующих значений `int`.

Дескриптор файла `FILE*` — это ссылка на файл, которым управляет операционная система. *Дескриптор* — это непрозрачная абстрактная ссылка на некоторый ресурс в операционной системе. Функция `fopen` принимает два аргумента: `имя-файла` ❷ — это путь к файлу, который нужно открыть, а `режим` ❸ — это один из шести параметров, показанных в таблице 11.2.

Файл должен быть закрыт вручную с помощью `fclose` после завершения его использования. Неспособность закрыть дескрипторы файлов является распространенным источником утечек ресурсов, например:

```
int fclose(FILE* file);
```

Таблица 11.2. Опции для всех шести режимов `fopen`

Строка	Операции	Файл существует:	Файл не существует:	Примечания
`r`	Чтение		Провал `fopen`	
`w`	Запись	Перезаписать	Создать файл	Если файл существует, все содержимое отбрасывается
`a`	Добавление		Создать файл	Запись всегда в конец файла
`r+`	Чтение/запись		Провал `fopen`	
`w+`	Чтение/запись	Перезаписать	Создать файл	Если файл существует, все содержимое отбрасывается
`a+`	Чтение/запись		Создать файл	Запись всегда в конец файла

Для записи в файл можно использовать функцию `fprintf`, которая похожа на `printf`, но выводит результат в файл вместо консоли. Функция `fprintf` имеет идентичное использование с `printf`, за исключением того, что в качестве первого аргумента перед строкой формата предоставляется дескриптор файла:

```
int❶ fprintf(FILE* file❷, const char* format_string❸, ...❹);
```

В случае успеха `fprintf` возвращает количество символов ❶, записанных в открытый файл ❷. Строка `format_string` такая же, как строка формата для `printf` ❸, так же, как и вариативные аргументы ❹.

Можно использовать `std::unique_ptr` для `FILE`. Очевидно, что не нужно вызывать `delete` для дескриптора файла `FILE*` в момент закрытия файла. Вместо этого нужно закрыть файл с помощью `fclose`. Поскольку `fclose` является функционально-подобным объектом, принимающим `FILE*`, он является подходящим удалителем.

Программа в листинге 11.13 записывает строку `HELLO DAVE` в файл `HAL9000` и использует уникальный указатель для управления ресурсами открытого файла.

Листинг 11.13. Программа, использующая std::unique_ptr и пользовательский удалитель для управления дескриптором файла

```
#include <cstdio>
#include <memory>

using FileGuard = std::unique_ptr<FILE, int(*)(FILE*)>; ❶

void say_hello(FileGuard file❷) {
  fprintf(file.get(), "HELLO DAVE"); ❸
}

int main() {
  auto file = fopen("HAL9000", "w"); ❹
  if (!file) return errno; ❺
  FileGuard file_guard{ file, fclose }; ❻
```

```
    // Открытие файла
    say_hello(std::move(file_guard)); ❼
    // Закрытие файла
    return 0;
}
```

Этот листинг создает псевдоним типа `FileGuard` ❶ для краткости. (Обратите внимание, что тип удалителя соответствует типу `fclose`.) Далее следует функция `say_hello`, которая принимает `FileGuard` по значению ❷. В пределах `say_hello` используется `fprintf` для вывода `HELLO DAVE` в `file` ❸. Поскольку время жизни файла привязано к `say_hello`, файл закрывается после возврата `say_hello`. В `main` файл `HAL9000` открывается в режиме `w`, который создаст или перезапишет файл, и сохраняется дескриптор обычного файла `FILE*` в `file` ❹. Вы проверяете, имеет ли файл значение `nullptr`, что указывает на ошибку, и возвращаете результат с ошибкой `errno`, если `HAL9000` не может быть открыт ❺. `FileGuard` создается при передаче файла дескриптора и пользовательского удалителя `fclose` ❻. На этом этапе файл открыт, и благодаря его собственному удалителю `file_guard` автоматически управляет временем жизни файла.

Чтобы вызвать `say_hello`, нужно передать право собственности на эту функцию (потому что она принимает `FileGuard` по значению) ❼. Вспомните из «Категорий значений», что такие переменные, как `file_guard`, являются `l`-значениями. Это означает, что нужно переместить ее в `say_hello` с помощью `std::move`, который записывает `HELLODAVE` в файл. Если опустить `std::move`, компилятор попытается скопировать его в `say_hello`. Поскольку `unique_ptr` имеет конструктор удаленных копий, это приведет к ошибке компилятора.

Когда `say_hello` возвращает результат, его аргумент `FileGuard` уничтожается и пользовательский удалитель вызывает `fclose` для дескриптора файла. По сути, утечка дескриптора файла невозможна. Он связан со временем существования `FileGuard`.

Неполный список поддерживаемых операций

Таблица 11.3 перечисляет все поддерживаемые операции `std::unique_ptr`. В этой таблице `ptr` означает обычный указатель, `u_ptr` — уникальный указатель, а `del` — удалитель.

Таблица 11.3. Все поддерживаемые операции `std::unique_ptr`

Операция	Примечания
`unique_ptr{ }` или `unique_ptr{ nullptr }`	Создает пустой уникальный указатель с удалителем `std::default_delete<...>`
`unique_ptr{ ptr }`	Создает уникальный указатель, владеющий динамическим объектом, на который указывает `ptr`. Использует удалитель `std::default _delete<...>`

Продолжение ⇩

Таблица 11.3 *(продолжение)*

Операция	Примечания
`unique_ptr{ ptr, del }`	Создает уникальный указатель, владеющий динамическим объектом, на который указывает `ptr`. Использует `del` в качестве удалителя
`unique_ptr{ move(u_ptr) }`	Создает уникальный указатель, владеющий динамическим объектом, на который указывает уникальный указатель `u_ptr`. Переносит владение из `u_ptr` во вновь созданный уникальный указатель. Также перемещает удалитель `u_ptr`
`~unique_ptr()`	Вызывает удалитель принадлежащего объекта, если полный
`u_ptr1 = move(u_ptr2)`	Передает право собственности на принадлежащий объект и удалитель из `u_ptr2` в `u_ptr1`. Уничтожает принадлежащий объект, если полный
`u_ptr1.swap(u_ptr2)`	Обменивается собственными объектами и удалителями между `u_ptr1` и `u_ptr2`
`swap(u_ptr1, u_ptr2)`	Свободная функция, идентичная методу `swap`
`u_ptr.reset()`	Если полный, вызывает удалитель для объекта, принадлежащего `u_ptr`
`u_ptr.reset(ptr)`	Удаляет принадлежащий объект; затем вступает во владение `ptr`
`ptr = u_ptr.release()`	Возвращает обычный указатель `ptr`; `u_ptr` становится пустым. Удалитель не вызывается
`ptr = u_ptr.get()`	Возвращает обычный указатель `ptr`; `u_ptr` сохраняет право владения
`*u_ptr`	Оператор разыменования на принадлежащем объекте
`u_ptr->`	Оператор разыменования члена на принадлежащем объекте
`u_ptr[index]`	Ссылка на элемент по индексу (только для массивов)
`bool{ u_ptr }`	Преобразование `bool`: `true`, если полный, `false`, если пустой

`u_ptr1 == u_ptr2`
`u_ptr1 != u_ptr2`
`u_ptr1 > u_ptr2`
`u_ptr1 >= u_ptr2`
`u_ptr1 < u_ptr2`
`u_ptr1 <= u_ptr2`
	Операторы сравнения; эквивалентны вычислению операторов сравнения на обычных указателях

`u_ptr.get_deleter()`	Возвращает ссылку на удалитель

Общие указатели

Общий указатель имеет передаваемое неисключительное владение одним динамическим объектом. Можно перемещать общие указатели, что делает их передаваемыми, и можно копировать их, что делает их владение неисключительным.

Неисключительное владение означает, что `shared_ptr` проверяет, владеют ли какие-либо другие объекты `shared_ptr` объектом перед его уничтожением. Таким образом, последний владелец освобождает принадлежащий объект.

У stdlib есть заголовок `std::shared_ptr`, доступный в заголовке `<memory>`, и Boost имеет `boost::shared_ptr`, доступный в заголовке `<boost/smart_ptr/shared_ptr.hpp>`. Здесь будет использоваться версия stdlib.

ПРИМЕЧАНИЕ

И stdlib, и Boost `shared_ptr`, по сути, идентичны, за заметным исключением, что общий указатель Boost не поддерживает массивы и требует от использования класса `boost::shared_array` в `<boost/smart_ptr/shared_array.hpp>`. Boost предлагает общий указатель по устаревшим причинам, но стоит использовать общий указатель stdlib.

Создание

Указатель `std::shared_ptr` поддерживает все те же конструкторы, что и `std::unique_ptr`. Конструктор по умолчанию выдает пустой общий указатель. Вместо этого, чтобы установить право собственности на динамический объект, можно передать указатель на конструктор `shared_ptr`, например:

```
std::shared_ptr<int> my_ptr{ new int{ 808 } };
```

Также существует следующая функция шаблона `std::make_shared`, которая перенаправляет аргументы в конструктор указанного типа:

```
auto my_ptr = std::make_shared<int>(808);
```

Как правило, нужно использовать `make_shared`. Для общих указателей требуется *блок управления*, который отслеживает несколько величин, включая количество общих владельцев. При использовании `make_shared` можно выделить блок управления и собственный динамический объект одновременно. Если сначала использовать оператор `new`, а затем выделить общий указатель, происходят два выделения вместо одного.

ПРИМЕЧАНИЕ

Иногда можно избежать использования `make_shared`. Например, если использовать `weak_ptr`, все равно понадобится управляющий блок, даже если можно освободить объект. В такой ситуации можно предпочесть два распределения.

Поскольку управляющий блок — это динамический объект, объектам `shared_ptr` иногда требуется выделить динамические объекты. Если нужно взять под контроль распределение `shared_ptr`, можно переопределить оператор `new`. Но это стрельба по воробьям из пушки. Более специализированный подход заключается в предоставлении необязательного параметра шаблона, называемого *типом распределителя*.

Определение распределителя

Распределитель отвечает за распределение, создание, уничтожение и освобождение объектов. Распределитель по умолчанию, `std::allocator`, является классом шаблона, определенным в заголовке `<memory>`. Распределитель по умолчанию выделяет память из динамического хранилища и принимает параметр шаблона. (Вы узнаете о настройке этого поведения с помощью пользовательского распределителя в разделе «Распределители»на с. 439.)

И конструктор `shared_ptr`, и `make_shared` имеют параметр шаблона типа распределителя, составляющий три общих параметра шаблона: тип объекта, на который указывает указатель, тип удалителя и тип распределителя. По сложным причинам нужно всего лишь объявить параметр *типа объекта, на который направлен указатель*. Можно представить другие типы параметров как выведенные из указанного типа.

Например, вот полностью разукрашенный вызов `make_shared`, включающий аргумент конструктора, пользовательский удалитель и явный `std::allocator`:

```
std::shared_ptr<int❶> sh_ptr{
  new int{ 10 }❷,
  [](int* x) { delete x; } ❸,
  std::allocator<int>{} ❹
};
```

Здесь указывается один параметр шаблона, `int`, для заданного типа ❶. В первом аргументе выделяется и инициализируется `int` ❷. Далее — пользовательский удалитель ❸, а в качестве третьего аргумента передается `std::allocator` ❹.

По техническим причинам нельзя использовать пользовательский удалитель или пользовательский распределитель с `make_shared`. Если нужен пользовательский распределитель, можно использовать сестринскую функцию `make_shared`, которая является `std::allocate_shared`. Она принимает в качестве первого аргумента распределитель и перенаправляет оставшиеся аргументы в конструктор принадлежащего объекта:

```
auto sh_ptr = std::allocate_shared<int❶>(std::allocator<int>{}❷, 10❸);
```

Как и в случае с `make_shared`, собственный тип указывается в качестве параметра шаблона ❶, но распределитель передается в качестве первого аргумента ❷. Остальные аргументы направляются в конструктор `int` ❸.

ПРИМЕЧАНИЕ

Пользовательский удалитель с `make_shared` нельзя использовать по двум причинам. Во-первых, `make_shared` использует `new` для выделения пространства принадлежащему объекту и управляющему блоку. Подходящим удалителем для `new` является `delete`, поэтому, как правило, пользовательский удалитель не подходит. Во-вторых, пользовательский удалитель обычно не может знать, как обращаться с блоком управления только принадлежащего ему объекта.

Невозможно указать пользовательский удалитель с помощью `make_shared` или `allocate_shared`. Если нужно использовать пользовательский удалитель с общими указателями, стоит использовать один из соответствующих конструкторов `shared_ptr` напрямую.

Поддерживаемые операции

`std::shared_ptr` поддерживает все операции, которые поддерживают `std::.unique_ptr` и `boost::scoped_ptr`. Можно использовать следующий псевдоним типа в качестве замены для `ScopedOathbreaker` в листингах с 11.1 по 11.7 и `UniqueOathbreakers` из листингов с 11.10 по 11.13:

```
using SharedOathbreakers = std::shared_ptr<DeadMenOfDunharrow>;
```

Основное функциональное различие между общим указателем и уникальным указателем заключается в том, что общие указатели можно копировать.

Переносимое и неисключительное владение

Общие указатели могут передаваться (их *можно* перемещать), и они имеют неисключительное право собственности (их *можно* копировать). Листинг 11.10, показывающий семантику переноса уникального указателя, работает так же для общего указателя.

Из листинга 11.14 видно, что совместно используемые указатели также поддерживают семантику копирования.

Листинг 11.14. std::shared_ptr поддерживает копирование

```
TEST_CASE("SharedPtr can be used in copy") {
  auto aragorn = std::make_shared<DeadMenOfDunharrow>();
  SECTION("construction") {
    auto son_of_arathorn{ aragorn }; ❶
    REQUIRE(DeadMenOfDunharrow::oaths_to_fulfill == 1); ❷
  }
  SECTION("assignment") {
    SharedOathbreakers son_of_arathorn; ❸
    son_of_arathorn = aragorn; ❹
    REQUIRE(DeadMenOfDunharrow::oaths_to_fulfill == 1); ❺
  }
```

```
    SECTION("assignment, and original gets discarded") {
      auto son_of_arathorn = std::make_shared<DeadMenOfDunharrow>(); ❻
      REQUIRE(DeadMenOfDunharrow::oaths_to_fulfill == 2); ❼
      son_of_arathorn = aragorn; ❽
      REQUIRE(DeadMenOfDunharrow::oaths_to_fulfill == 1); ❾
    }
  }
```

После создания общего указателя `aragorn` появляются три теста. Первый тест показывает, что конструктор копирования, который используется для создания `son_of_arathorn` ❶, делит владение с тем же `DeadMenOfDunharrow` ❷.

Во втором тесте создается пустой общий указатель `son_of_arathorn` ❸, а затем показывается, что присваивание копии ❹ также не меняет число `DeadMenOfDunharrow` ❺.

Третий тест показывает, что при создании полного общего указателя `son_of_arathorn` ❻ число `DeadMenOfDunharrow` увеличивается до 2 ❼. При копировании присваивания `aragorn` в `son_of_arathorn` ❽ `son_of_arathorn` удаляет его `DeadMenOfDunharrow`, потому что он имеет исключительное владение. Затем он увеличивает счетчик ссылок `DeadMenOfDunharrow`, принадлежащих `aragorn`. Поскольку оба общих указателя имеют один и тот же `DeadMenOfDunharrow`, значение `oaths_to_fulfill` уменьшается с 2 до 1 ❾.

Общие массивы

Общий массив — это общий указатель, который владеет динамическим массивом и поддерживает оператор `[]`. Работает так же, как уникальный массив, за исключением того, что он имеет неисключительное владение.

Удалители

Для общих указателей удалители работают так же, как и для уникальных, за исключением того, что не нужно указывать параметр шаблона с типом удалителя.

Просто передайте удалитель как второй аргумент конструктора. Например, чтобы преобразовать листинг 11.13 для использования общего указателя, просто добавьте псевдоним следующего типа:

```
using FileGuard = std::shared_ptr<FILE>;
```

Теперь вы управляете дескрипторами файлов `FILE*` с общим владением.

Неполный список поддерживаемых операций

В таблице 11.4 приведен в основном полный список поддерживаемых конструкторов `shared_ptr`. В этой таблице `ptr` означает обычный указатель, `sh_ptr` — общий указатель, `u_ptr` — уникальный указатель, `del` — удалитель, а `alc` — распределитель.

Таблица 11.4. Все поддерживаемые конструкторы `std::shared_ptr`

Операция	Примечания
`shared_ptr{ }` или `shared_ptr{ nullptr }`	Создает пустой общий указатель с `std::default_delete<T>` и `std::allocator<T>`
`shared_ptr{ ptr, [del], [alc] }`	Создает общий указатель, владеющий динамическим объектом, на который указывает `ptr`. Использует `std::default_delete<T>` и `std::allocator<T>` по умолчанию; в противном случае `del` используется как удалитель, `alc` — как распределитель, если они имеются
`shared_ptr{ sh_ptr }`	Создает общий указатель, владеющий динамическим объектом, на который указывает общий указатель `sh_ptr`. Копирует владение из `sh_ptr` во вновь созданный общий указатель. Также копирует удалитель и распределитель `sh_ptr`
`shared_ptr{ sh_ptr, ptr }`	Конструктор псевдонимов: результирующий общий указатель содержит неуправляемую ссылку на `ptr`, но участвует в подсчете ссылок `sh_ptr`
`shared_ptr{ move (sh_ptr) }`	Создает общий указатель, владеющий динамическим объектом, на который указывает общий указатель `sh_ptr`. Переносит владение из `sh_ptr` во вновь созданный общий указатель. Также перемещает удалитель `sh_ptr`
`shared_ptr{ move (u_ptr) }`	Создает общий указатель, владеющий динамическим объектом, на который указывает уникальный указатель `u_ptr`. Переносит владение из `u_ptr` во вновь созданный общий указатель. Также перемещает удалитель `u_ptr`

В таблице 11.5 приведен список большинства поддерживаемых операций `std::shared_ptr`. В этой таблице `ptr` означает обычный указатель, `sh_ptr` — общий указатель, `u_ptr` — уникальный указатель, `del` — удалитель, а `alc` — распределитель.

Таблица 11.5. Большинство поддерживаемых операций `std::shared_ptr`

Операция	Примечания
`~shared_ptr()`	Вызывает удалитель принадлежащего объекта, если других владельцев не существует
`sh_ptr1 = sh_ptr2`	Копирует владение принадлежащим объектом и удалитель из `sh_ptr2` в `sh_ptr1`. Увеличивает количество владельцев на 1. Уничтожает находящийся в собственности объект, если нет другого владельца
`sh_ptr = move(u_ptr)`	Передает право собственности на принадлежащий объект и удалитель из `u_ptr` в `sh_ptr`. Уничтожает находящийся в собственности объект, если других владельцев не существует
`sh_ptr1 = move (sh_ptr2)`	Передает право собственности на принадлежащий объект и удалитель из `sh_ptr2` в `sh_ptr1`. Уничтожает находящийся в собственности объект, если других владельцев не существует

Продолжение ⇩

Таблица 11.5 *(продолжение)*

Операция	Примечания
`sh_ptr1.swap(sh_ptr2)`	Обменивается принадлежащими объектами и удалителями между `sh_ptr1` и `sh_ptr2`
`swap(sh_ptr1, sh_ptr2)`	Свободная функция, идентичная методу `swap`
`sh_ptr.reset()`	Если полный, вызывает удалитель для объекта, принадлежащего `sh_ptr`, если других владельцев не существует
`sh_ptr.reset(ptr, [del], [alc])`	Удаляет принадлежащий в настоящее время объект, если других владельцев не существует; затем вступает во владение `ptr`. При желании можно указать `delter del` и `allocator alc`. По умолчанию это `std::default_delete<T>` и `std::allocator<T>`
`ptr = sh_ptr.get()`	Возвращает обычный указатель `ptr`; `sh_ptr` сохраняет право владения
`*sh_ptr`	Оператор разыменования на принадлежащем объекте
`sh_ptr->`	Оператор разыменования члена на принадлежащем объекте
`sh_ptr.use_count()`	Указывает общее количество общих указателей, владеющих принадлежащим объектом; ноль, если пустой
`sh_ptr[index]`	Возвращает элемент по индексу `index` (только для массивов)
`bool{ sh_ptr }`	Преобразование `bool`: `true`, если полный, `false`, если пустой

`sh_ptr1 == sh_ptr2`
`sh_ptr1 != sh_ptr2`
`sh_ptr1 > sh_ptr2`
`sh_ptr1 >= sh_ptr2`
`sh_ptr1 < sh_ptr2`
`sh_ptr1 <= sh_ptr2`
	Операторы сравнения; эквивалентны вычислению операторов сравнения на обычных указателях

`sh_ptr.get_deleter()`	Возвращает ссылку на удалитель

Слабые указатели

Слабый указатель — это особый вид умного указателя, который не имеет права собственности на объект, на который он ссылается. Слабые указатели позволяют отслеживать объект и преобразовывать слабый указатель в общий указатель, *только если отслеживаемый объект все еще существует*. Это позволяет генерировать временное владение объектом. Как и общие указатели, слабые указатели являются переносимыми и копируемыми.

Обычно для слабых указателей используются *кэши*. В программной инженерии кэш — это структура данных, которая временно хранит данные, чтобы их можно

было быстрее извлекать. Кэш может содержать слабые указатели на объекты, поэтому они уничтожаются, как только все другие владельцы освобождают их. Периодически кэш может сканировать свои сохраненные слабые указатели и обрезать их без других владельцев.

В stdlib есть `std::weak_ptr`, а в Boost — `boost::weak_ptr`. Они по сути идентичны и предназначены только для использования с их соответствующими общими указателями, `std::shared_ptr` и `boost::shared_ptr`.

Создание

Конструкторы слабых указателей полностью отличаются от уникальных и общих и ограниченных указателей, поскольку слабые указатели не имеют непосредственных собственных динамических объектов. Конструктор по умолчанию создает пустой слабый указатель. Чтобы создать слабый указатель, который отслеживает динамический объект, нужно создать его, используя общий указатель или другой слабый указатель.

Например, передает общий указатель в конструктор слабого указателя следующее:

```
auto sp = std::make_shared<int>(808);
std::weak_ptr<int> wp{ sp };
```

Теперь слабый указатель `wp` будет отслеживать объект, принадлежащий общему указателю `sp`.

Получение временного владения

Слабые указатели вызывают свой метод `lock` для временного владения отслеживаемым объектом. Метод `lock` всегда создает общий указатель. Если отслеживаемый объект жив, возвращенный общий указатель владеет отслеживаемым объектом. Если отслеживаемый объект больше не является живым, возвращенный общий указатель является пустым. Рассмотрите пример в листинге 11.15.

В первом тесте создается общий указатель `aragorn` ❶ с сообщением. Затем создается слабый указатель `legolas` с помощью `aragorn` ❷. Это устанавливает `legolas` для отслеживания динамического объекта, принадлежащего `aragorn`. При вызове `lock` слабого указателя ❸ `aragorn` все еще жив, поэтому в результате получается общий указатель `sh_ptr`, которому также принадлежит тот же `DeadMenOfDunharrow`. Вы подтверждаете это, утверждая, что `message` то же самое ❹ и что *счетчик использования* равен 2 ❺.

Во втором тесте также создается общий указатель `aragorn` ❻, но на этот раз используется оператор присваивания ❼, поэтому ранее пустой `legolas` слабого указателя теперь отслеживает динамический объект, принадлежащий `aragorn`. Затем `aragorn` выпадает из блока и уничтожается. Это оставляет `legolas`, отслеживающего уничтоженный объект. При вызове `lock` в этой точке ❽ вы получаете пустой общий указатель ❾.

Листинг 11.15. std::weak_ptr предоставляет метод lock для получения временного владения

```
TEST_CASE("WeakPtr lock() yields") {
  auto message = "The way is shut.";
  SECTION("a shared pointer when tracked object is alive") {
    auto aragorn = std::make_shared<DeadMenOfDunharrow>(message); ❶
    std::weak_ptr<DeadMenOfDunharrow> legolas{ aragorn }; ❷
    auto sh_ptr = legolas.lock(); ❸
    REQUIRE(sh_ptr->message == message); ❹
    REQUIRE(sh_ptr.use_count() == 2); ❺
  }
  SECTION("empty when shared pointer empty") {
    std::weak_ptr<DeadMenOfDunharrow> legolas;
    {
      auto aragorn = std::make_shared<DeadMenOfDunharrow>(message); ❻
      legolas = aragorn; ❼
    }
    auto sh_ptr = legolas.lock(); ❽
    REQUIRE(nullptr == sh_ptr); ❾
  }
}
```

Продвинутые шаблоны

В некоторых случаях расширенного использования общих указателей можно создать класс, который позволит экземплярам создавать общие указатели, ссылающиеся на самих себя. Шаблон класса `std::enable_shared_from_this` реализует это поведение. Все, что требуется с точки зрения пользователя — это наследовать от `enable_shared_from_this` в определении класса. Это предоставляют методы `shared_from_this` и `weak_from_this`, которые генерируют `shared_ptr` или `weak_ptr` со ссылкой на текущий объект. Это специальный случай, но если вы хотите увидеть больше деталей, обратитесь к [util.smartptr.enab].

Поддерживаемые операции

В таблице 11.6 перечислены большинство поддерживаемых операций со слабыми указателями. В этой таблице `w_ptr` означает слабый указатель, а `sh_ptr` — общий указатель.

Навязчивые указатели

Навязчивый указатель — это общий указатель на объект со встроенным счетчиком ссылок. Поскольку общие указатели обычно ведут подсчет ссылок, они не подходят для владения такими объектами. Boost предоставляет реализацию `boost::intrusive_ptr` в заголовке `<boost/smart_ptr/intrusive_ptr.hpp>`.

Таблица 11.6: Большинство поддерживаемых операций `std::shared_ptr`

Операция	Примечания
`weak_ptr{ }`	Создает пустой слабый указатель
`weak_ptr{ w_ptr }` или `weak_ptr{ sh_ptr }`	Отслеживает объект, на который ссылается слабый указатель `w_ptr` или общий указатель `sh_ptr`
`weak_ptr{ move(w_ptr) }`	Отслеживает объект, на который ссылается `w_ptr`; затем очищает `w_ptr`
`~weak_ptr()`	Не влияет на отслеживаемый объект
`w_ptr1 = sh_ptr` или `w_ptr1 = w_ptr2`	Заменяет отслеживаемый в данный момент объект на объект, принадлежащий `sh_ptr` или отслеживаемый `w_ptr2`
`w_ptr1 = move(w_ptr2)`	Заменяет текущий отслеживаемый объект объектом, отслеживаемым с помощью `w_ptr2`. Очищает `w_ptr2`
`sh_ptr = w_ptr.lock()`	Создает общий указатель `sh_ptr`, владеющий объектом, отслеживаемым `w_ptr`. Если отслеживаемый объект истек, `sh_ptr` пустой
`w_ptr1.swap(w_ptr2)`	Обменивается отслеживаемыми объектами между `w_ptr1` и `w_ptr2`
`swap(w_ptr1, w_ptr2)`	Свободная функция, идентичная методу `swap`
`w_ptr.reset()`	Очищает слабый указатель
`w_ptr.use_count()`	Указывает общее количество общих указателей, владеющих отслеживаемым объектом
`w_ptr.expired()`	Возвращает `true`, если время жизни отслеживаемого объекта истекло, `false`, если это не так
`sh_ptr.use_count()`	Указывает общее количество общих указателей, владеющих принадлежащим объектом; ноль, если пустой

Редкий случай, когда ситуация требует навязчивого указателя. Но иногда используются операционной системой или платформой, которые содержат встроенные ссылки. Например, в программировании Windows COM навязчивый указатель может быть очень полезен: COM-объекты, которые наследуются от интерфейса `IUnknown`, имеют методы `AddRef` и `Release`, которые увеличивают и уменьшают встроенный счетчик ссылок (соответственно).

Каждый раз, когда создается `intrusive_ptr`, он вызывает функцию `intrusive_ptr_add_ref`. Когда `intrusive_ptr` уничтожается, он вызывает свободную функцию `intrusive_ptr_release`. Вы несете ответственность за освобождение соответствующих ресурсов в `intrusive_ptr_release`, когда количество ссылок падает до нуля. Чтобы использовать навязчивый `_ptr`, нужно предоставить подходящую реализацию этих функций.

В листинге 11.16 показаны навязчивые указатели с использованием класса `DeadMenOfDunharrow`. Рассмотрим реализации `intrusive_ptr_add_ref` и `intrusive_ptr_release` в этом листинге.

Листинг 11.16. Реализации intrusive_ptr_add_ref и intrusive_ptr_release

```
#include <boost/smart_ptr/intrusive_ptr.hpp>

using IntrusivePtr = boost::intrusive_ptr<DeadMenOfDunharrow>; ❶
size_t ref_count{}; ❷

void intrusive_ptr_add_ref(DeadMenOfDunharrow* d) {
  ref_count++; ❸
}

  void intrusive_ptr_release(DeadMenOfDunharrow* d) {
  ref_count--; ❹
  if (ref_count == 0) delete d; ❺
}
```

Использование псевдонима типа `IntrusivePtr` позволяет сэкономить время при наборе кода ❶. Затем объявляется `ref_count` со статической длительностью хранения ❷. Эта переменная отслеживает количество живых навязчивых указателей. В `intrusive_ptr_add_ref` `ref_count` увеличивается ❸. В `intrusive_ptr_release` `ref_count` уменьшается ❹. Когда `ref_count` падает до нуля, аргумент `DeadMenOfDunharrow` удаляется ❺.

ПРИМЕЧАНИЕ

При использовании установки в листинге 11.16 крайне важно использовать только один динамический объект `DeadMenOfDunharrow` с навязчивыми указателями. Подход `ref_count` будет правильно отслеживать только один объект. Если есть несколько динамических объектов, принадлежащих разным навязчивым указателям, `ref_count` станет недействительным, и получится неправильное поведение удаления ❺.

В листинге 11.17 показано, как использовать установку из листинга 11.16 с навязчивыми указателями.

Листинг 11.17. Использование boost::intrusive_ptr

```
TEST_CASE("IntrusivePtr uses an embedded reference counter.") {
  REQUIRE(ref_count == 0); ❶
  IntrusivePtr aragorn{ new DeadMenOfDunharrow{} }; ❷
  REQUIRE(ref_count == 1); ❸
  {
    IntrusivePtr legolas{ aragorn }; ❹
    REQUIRE(ref_count == 2); ❺
  }
  REQUIRE(DeadMenOfDunharrow::oaths_to_fulfill == 1); ❻
}
```

Этот тест начинается с проверки того, что `ref_count` равен нулю ❶. Затем создается навязчивый указатель, передавая динамически выделенный `DeadMenOfDunharrow` ❷.

Это увеличивает `ref_count` до 1, потому что создание навязчивого указателя вызывает `intrusive_ptr_add_ref` ❸. В пределах блока создается другой навязчивый указатель `legolas`, который делит владение с `aragorn` ❹. Это увеличивает `ref_count` до 2 ❺, потому что создание навязчивого указателя вызывает `intrusive_ptr_add_ref`. Когда `legolas` выпадает из области видимости блока, он уничтожается, вызывая `intrusive_ptr_release`. Это уменьшает `ref_count` до 1, но не заставляет принадлежащий объект удалять ❻.

Обзор вариантов умных указателей

Таблица 11.7 суммирует все опции умного указателя, доступные для использования в stdlib и Boost.

Таблица 11.7. Умные указатели в stdlib и Boost

Имя типа	Заголовок stdlib	Заголовок Boost	Переносимое/перемещаемое владение	Копируемое/неисключительное владение
`scoped_ptr`		`<boost/smart_ptr/scoped_ptr.hpp>`		
`scoped_array`		`<boost/smart_ptr/scoped_array.hpp>`		
`unique_ptr`	`<memory>`		✓	
`shared_ptr`	`<memory>`	`<boost/smart_ptr/shared_ptr.hpp>`	✓	✓
`shared_array`		`<boost/smart_ptr/shared_array.hpp>`	✓	✓
`weak_ptr`	`<memory>`	`<boost/smart_ptr/weak_ptr.hpp>`	✓	✓
`intrusive_ptr`		`<boost/smart_ptr/intrusive_ptr.hpp>`	✓	✓

Распределители

Распределители — это объекты низкого уровня, которые обслуживают запросы памяти. Библиотеки stdlib и Boost позволяют предоставлять распределители для настройки того, как библиотека выделяет динамическую память.

В большинстве случаев стандартного распределителя `std::allocate` вполне достаточно. Он выделяет память с помощью оператора `new(size_t)`, который выделяет необработанную память из свободного хранилища, также известного как куча. Он освобождает память с помощью оператора `delete(void*)`, который освобождает

необработанную память из свободного хранилища. (Вспомните из «Перегрузки оператора new», что оператор `new` и оператор `delete` определены в заголовке `<new>`.)

В некоторых разработках, таких как игры, высокочастотная торговля, научный анализ и встроенные приложения, расходы на память и вычислительные затраты, связанные с операциями бесплатного хранилища по умолчанию, недопустимы. В таких разработках относительно легко реализовать свой собственный распределитель. Обратите внимание, что действительно не следует реализовывать пользовательский распределитель, если вы не провели некоторое тестирование производительности, которое показывает, что распределитель по умолчанию является узким местом. Идея пользовательского распределителя состоит в том, что вы знаете намного больше о своей конкретной программе, чем разработчики модели распределителя по умолчанию, поэтому вы можете внести улучшения, которые повысят производительность выделения.

Как минимум необходимо предоставить класс шаблона со следующими характеристиками, чтобы он работал в качестве распределителя:

- соответствующий конструктор по умолчанию;
- член `value_type`, соответствующий параметру шаблона;
- конструктор шаблона, который может копировать внутреннее состояние распределителя при обработке с изменением в `value_type`;
- метод `allocate`;
- метод `deallocate`;
- `operator==`и `operator!=`.

Класс `MyAllocator` в листинге 11.18 реализует простой учебный вариант: `std::allocate`, который отслеживает, сколько выделений и освобождений было сделано.

Листинг 11.18. Класс MyAllocator, смоделированный после std::allocate

```
#include <new>

static size_t n_allocated, n_deallocated;

template <typename T>
struct MyAllocator {
  using value_type = T; ❶
  MyAllocator() noexcept{ } ❷
  template <typename U>
  MyAllocator(const MyAllocator<U>&) noexcept { } ❸
  T* allocate(size_t n) { ❹
    auto p = operator new(sizeof(T) * n);
    ++n_allocated;
    return static_cast<T*>(p);
  }
  void deallocate(T* p, size_t n) { ❺
    operator delete(p);
```

```
    ++n_deallocated;
  }
};

template <typename T1, typename T2>
bool operator==(const MyAllocator<T1>&, const MyAllocator<T2>&) {
  return true; ❻
}
template <typename T1, typename T2>
bool operator!=(const MyAllocator<T1>&, const MyAllocator<T2>&) {
  return false; ❼
}
```

Сначала объявляется псевдоним типа `value_type` для `T`, одного из требований для реализации распределителя ❶. Далее идет конструктор по умолчанию ❷ и конструктор шаблона ❸. Оба они пусты, потому что у распределителя нет состояния для передачи.

Метод `allocate` ❹ моделирует `std::allocate`, выделяя необходимое количество байтов `sizeof(T) * n`, используя оператор `new`. Затем он увеличивает статическую переменную `n_allocated`, чтобы можно было отслеживать количество выделений для целей тестирования. Затем метод `allocate` возвращает указатель на вновь выделенную память после приведения `void *` к соответствующему типу указателя.

Метод `deallocate` ❺ также моделирует `std::allocate`, вызывая оператор `delete`. В качестве аналогии с распределением он увеличивает статическую переменную `n_deallocated` для тестирования и возвращает результат.

Последняя задача — реализовать `operator==` и `operator!=`, принимающие новый шаблон класса. Поскольку распределитель не имеет состояния, любой экземпляр является таким же, как и любой другой экземпляр, поэтому `operator==` возвращает `true` ❻, и `operator!=` возвращает `false`❼.

ПРИМЕЧАНИЕ

Листинг 11.18 является обучающим и фактически не делает распределение более эффективным. Он просто оборачивает вызов `new` и `delete`.

Пока что единственный известный класс, использующий распределитель, это `std::shared_ptr`. Рассмотрим, как в листинге 11.19 `MyAllocator` используется совместно с `std::allocate`.

Вы создаете экземпляр `MyAllocator` с именем `alloc` ❶. Внутри блока `alloc` передается в качестве первого аргумента `allocate_shared` ❷, который создает общий указатель `aragorn`, содержащий пользовательский `message` ❸. Затем вы подтверждаете, что `aragorn` содержит правильный `message` ❹, `n_allocated` равно 1 ❺, а `n_deallocated` равно 0 ❻.

После того как `aragorn` выпадает из области видимости блока и уничтожается, вы проверяете, что `n_allocated` все еще равен 1 ❼, а `n_deallocated` теперь равен 1 ❽.

Листинг 11.19. Использование MyAllocator с std::shared_ptr

```
TEST_CASE("Allocator") {
  auto message = "The way is shut.";
  MyAllocator<DeadMenOfDunharrow> alloc; ❶
  {
    auto aragorn = std::allocate_shared<DeadMenOfDunharrow>(alloc❷,
                                                            message❸);
    REQUIRE(aragorn->message == message); ❹
    REQUIRE(n_allocated == 1); ❺
    REQUIRE(n_deallocated == 0); ❻
  }
  REQUIRE(n_allocated == 1); ❼
  REQUIRE(n_deallocated == 1); ❽
}
```

ПРИМЕЧАНИЕ

Поскольку распределители обрабатывают низкоуровневые детали, можно действительно запутаться, определяя их поведение. См. [Allocator.requirements] в стандарте ISO C++17, чтобы разобраться более полно.

Итоги

Умные указатели управляют динамическими объектами через RAII, и можно предоставить распределители для настройки динамического выделения памяти. В зависимости от того, какой умный указатель вы выберете, можно закодировать различные шаблоны владения динамическим объектом.

Упражнения

11.1. Переопределите листинг 11.13, чтобы использовать `std::shared_ptr` вместо `std::unique_ptr`. Обратите внимание, что, хотя вы изменили требования к владению с исключительных на неисключительные, право владения функцией `say_hello` по-прежнему передается.

11.2. Удалите `std::move` из вызова `say_hello`. Затем сделайте дополнительный вызов `say_hello`. Обратите внимание, что право владения `file_guard` больше не передается `say_hello`. Это позволяет провести множественные вызовы.

11.3. Реализуйте класс `Hal`, который принимает `std::shared_ptr<FILE>` в конструкторе. В деструкторе `Hal` напишите фразу `Stop Dave` в дескриптор файла, хранящегося в общем указателе. Реализуйте функцию `write_status`, которая записывает фразу`I'm completely operational.` в дескриптор файла. Вот объявление класса, с которым можно работать:

```
struct Hal {
  Hal(std::shared_ptr<FILE> file);
  ~Hal();
  void write_status();
  std::shared_ptr<FILE> file;
};
```

11.4. Создайте несколько экземпляров `Hal` и вызовите для них `write_status`. Обратите внимание, что не нужно отслеживать количество открытых экземпляров `Hal`: управление файлами осуществляется с помощью модели общего владения общим указателем.

Что еще почитать?

- Международный стандарт ИСО/МЭК (2017) — Язык программирования C++ (Международная организация по стандартизации; Женева, Швейцария; isocpp.org/std/the-standard/)
- «Язык программирования C++», 4-е издание, Бьёрн Страуструп (Бином, 2011)
- «The Boost C++ Libraries», 2nd Edition, Boris Schäling (XML Press, 2014)
- «Стандартная библиотека C++. Справочное руководство», 2-е издание, Николай М. Джосаттис (Вильямс, 2017)

12 Утилиты

«Понимаешь, в мире полно существ, которые намного сильнее нас. Но если знаешь, как оседлать волну, далеко можно зайти», — говорит Ворон.
«Вот именно. В самую точку».

Нил Стивенсон, «Снежная катастрофа»

Библиотеки stdlib и Boost предоставляют множество типов, классов и функций, которые удовлетворяют общим требованиям программирования. Вместе эта пестрая коллекция инструментов называется *утилитами*. Помимо своей небольшой, несложной и целенаправленной природы утилиты различаются по функциональности.

В этой главе вы узнаете о нескольких простых структурах данных, которые обрабатывают многие рутинные ситуации, когда нужны объекты, содержащие другие объекты. Далее поговорим о датах и времени, а также об охвате нескольких положений о кодировании календарей и часов и об измерении прошедшего времени. Глава завершается множеством доступных числовых и математических инструментов.

ПРИМЕЧАНИЕ

Обсуждение дат/времени и чисел/математики будет представлять большой интерес для одних читателей и почти нулевой для других. Если вы принадлежите ко второй категории, не стесняйтесь просматривать эти разделы по диагонали.

Структуры данных

Между тем библиотеки stdlib и Boost предоставляют почтенную коллекцию полезных структур данных. *Структура данных* — это тип, который хранит объекты

и разрешает некоторый набор операций над этими сохраненными объектами. Не существует серебряной пули компилятора, которая заставляет работать служебные структуры данных в этом разделе; можно было бы реализовать свои собственные версии с достаточным временем и усилиями. Но зачем изобретать велосипед?

tribool

Tribool — это тип, подобный `bool`, который поддерживает три состояния, а не два: истина, ложь и неопределенность. Boost предлагает `boost::logic::tribool` в заголовке `<boost/logic/tribool.hpp>`. Листинг 12.1 демонстрирует, как инициализировать `tribool` в Boost, используя `true`, `false` и тип `boost::logic::indeterminate`.

Листинг 12.1. Инициализация tribool в Boost

```
#include <boost/logic/tribool.hpp>

using boost::logic::indeterminate; ❶
boost::logic::tribool t = true❷, f = false❸, i = indeterminate❹;
```

Для удобства объявление использования извлекает `indeterminate` из `boost::logic` ❶. Затем `tribool` инициализируется равным `true` ❷, `f` равным `false` ❸, а `i` равным `indeterminate` ❹.

Класс `tribool` неявно преобразуется в `bool`. Если `tribool` имеет значение `true`, он преобразуется в `true`; в противном случае — в `false`. Класс `tribool` также поддерживает `operator!`, который возвращает `true`, если `tribool` равен `false`; в противном случае возвращается `false`.

Наконец, `undeterminate` поддерживает `operator()`, который принимает один аргумент `tribool` и возвращает `true`, если этот аргумент `indeterminate`; в противном случае возвращается `false`.

В листинге 12.2 приведены примеры этих булевых преобразований.

Листинг 12.2. Преобразование tribool в bool

```
TEST_CASE("Boost tribool converts to bool") {
  REQUIRE(t); ❶
  REQUIRE_FALSE(f); ❷
  REQUIRE(!f); ❸
  REQUIRE_FALSE(!t); ❹
  REQUIRE(indeterminate(i)); ❺
  REQUIRE_FALSE(indeterminate(t)); ❻
}
```

Этот тест демонстрирует основные результаты преобразования `bool` ❶❷, `operator!` ❸❹ и `indeterminate` ❺❻.

Логические операции

Класс `tribool` поддерживает все логические операторы. Всякий раз, когда выражение `tribool` не содержит значения `indeterminate`, результат совпадает с эквивалентным

логическим выражением. Когда задействован неопределенный результат, результат может быть `indeterminate`, как показано в листинге 12.3.

Листинг 12.3. boost::tribool поддерживает логические операции

```
TEST_CASE("Boost Tribool supports Boolean operations") {
  auto t_or_f = t || f;
  REQUIRE(t_or_f); ❶
  REQUIRE(indeterminate(t && indeterminate)); ❷
  REQUIRE(indeterminate(f || indeterminate)); ❸
  REQUIRE(indeterminate(!i)); ❹
}
```

Поскольку ни `t`, ни `f` не являются `indeterminate`, `t || f` вычисляется так же, как и обычное логическое выражение, поэтому `t_or_f` имеет значение `true` ❶. Логические выражения со значением `indeterminate` могут быть `indeterminate`. Логические операторы И ❷, ИЛИ ❸ и НЕ ❹ вычисляются как `indeterminate`, если информации недостаточно.

Когда использовать tribool

Помимо описания жизненного статуса кота Шрёдингера, `tribool` можно использовать в условиях, когда операции могут занимать много времени. В таких разработках `tribool` может описать, была ли операция успешной. Значение `indeterminate` может моделировать, что операция еще не завершена.

Класс `tribool` обеспечивает аккуратные, лаконичные операторы `if`, как показано в листинге 12.4.

Листинг 12.4. Использование оператора if с tribool

```
TEST_CASE("Boost Tribool works nicely with if statements") {
  if (i) FAIL("Indeterminate is true."); ❶
  else if (!i) FAIL("Indeterminate is false.");❷
  else {} // OK, indeterminate ❸
}
```

Первое выражение ❶ вычисляется только в том случае, если значение `tribool` равно `true`, второе выражение ❷ вычисляется только в том случае, если оно равно `false`, а третье выполняется только в случае `indeterminate` ❸.

ПРИМЕЧАНИЕ

Простое упоминание `tribool` могло вызвать отвращение. Почему, спросите вы, нельзя просто использовать целое число, где 0 — ложь, 1 — истина, а любое другое значение — неопределенное? Да, так можно, но учтите, что тип `tribool` поддерживает все обычные логические операции при правильном распространении неопределенных значений. Опять же, зачем изобретать велосипед?

Неполный список поддерживаемых операций

В таблице 12.1 приведен список большинства поддерживаемых операций `boost::tribool`. В этой таблице `tb` означает `boost::tribool`.

Таблица 12.1. Большинство поддерживаемых операций `boost::tribool`

Операция	Примечания
`tribool{}` `tribool{ false }`	Создает `tribool` со значением `false`
`tribool{ true }`	Создает `tribool` со значением `true`
`tribool{ indeterminate }`	Создает `tribool` со значением `indeterminate`
`tb.safe_bool()`	Вычисляется как `true`, если `tb` равен `true`, иначе — `false`
`indeterminate(tb)`	Вычисляется как `true`, если `tb` равен `indeterminate`, иначе — `false`
`!tb`	Вычисляется как `true`, если `tb` равен `true`, иначе — `false`
`tb1 && tb2`	Вычисляется как `true`, если `tb1` и `tb2` равны `true`; вычисляется как `false`, если `tb1` или `tb2` равны `false`; иначе — `indeterminate`
`tb1 || tb2`	Вычисляется как `true`, если `tb1` или `tb2` равны `true`; вычисляется как `false`, если `tb1` и `tb2` равны `false`; иначе — `indeterminate`
`bool{ tb }`	Вычисляется как `true`, если `tb` равен `true`, иначе — `false`

optional

optional — это шаблон класса, который содержит значение, которое может присутствовать или не присутствовать. Основным вариантом использования параметра `optional` является тип возврата функции, которая может завершиться ошибкой. Вместо того чтобы генерировать исключение или возвращать несколько значений, функция может вместо этого возвращать параметр `optional`, который будет содержать значение, если функция выполнена успешно.

`std::option` содержится в stdlib в заголовке `<option>`, а в Boost имеется `boost::option` в заголовке `<boost/option.hpp>`.

Рассмотрите настройку в листинге 12.5. Функция `take` хочет вернуть экземпляр `TheMatrix` только при предоставлении `Pill::Blue`; в противном случае `take` возвращает `std::nullopt`, который является константой `stdlib`, предоставленной типом `std::optional` с неинициализированным состоянием.

Листинг 12.5. Функция take, возвращающая std::optional

```
#include <optional>

struct TheMatrix { ❶
  TheMatrix(int x) : iteration { x } { }
  const int iteration;
};

enum Pill { Red, Blue }; ❷

std::optional<TheMatrix>❸ take(Pill pill❹) {
  if(pill == Pill::Blue) return TheMatrix{ 6 }; ❺
  return std::nullopt; ❻
}
```

Тип `TheMatrix` принимает один аргумент конструктора `int` и сохраняет его в члене `iteration` ❶. `enum Pill` принимает значения `Red` и `Blue` ❷. Функция `take` возвращает `std::option<TheMatrix>` ❸ и принимает один аргумент `Pill` ❹. Если передать `Pill::Blue` функции `take`, она вернет экземпляр `TheMatrix` ❺; в противном случае возвращается `std::nullopt` ❻.

Сначала рассмотрим листинг 12.6, где принимается `blue pill` («синяя таблетка»).

Листинг 12.6. Тест, исследующий тип std::optionalc Pill::Blue

```
TEST_CASE("std::optional contains types") {
  if (auto matrix_opt = take(Pill::Blue)) { ❶
    REQUIRE(matrix_opt->iteration == 6); ❷
    auto& matrix = matrix_opt.value();
    REQUIRE(matrix.iteration == 6); ❸
  } else {
    FAIL("The optional evaluated to false.");
  }
}
```

Принимается «синяя таблетка», в результате чего появляется `std::optional`, содержащий инициализированный `TheMatrix`, поэтому условное выражение оператора `if` оценивается как `true` ❶. Листинг 12.6 также демонстрирует использование `operator->` ❷ и `value()` ❸ для доступа к внутреннему значению.

Что происходит, если принять «красную таблетку» (red pill)? Рассмотрим листинг 12.7.

Листинг 12.7. Тест, исследующий тип std::optionalc Pill::Red

```
TEST_CASE("std::optional can be empty") {
  auto matrix_opt = take(Pill::Red); ❶
  if (matrix_opt) FAIL("The Matrix is not empty."); ❷
  REQUIRE_FALSE(matrix_opt.has_value()); ❸
}
```

Принимается «красная таблетка» ❶, и полученный `matrix_opt` пуст. Это означает, что `matrix_opt` преобразуется в `false` ❷, а `has_value()` также возвращает `false` ❸.

Неполный список поддерживаемых операций

В таблице 12.2 приведен список большинства поддерживаемых операций `std::optional`. В этой таблице `opt` означает `std::optional<T>`, а `t` является объектом типа `T`.

Таблица 12.2. Большинство поддерживаемых операций `std::optional`

Операция	Примечания
`optional{}` `optional{std::nullopt}`	Создает пустой `optional`
`optional{ opt }`	Создает `optional` из `opt` с использованием конструктора копирования
`optional{ move(opt) }`	Создание `optional` из `opt` с использованием конструктора переноса, который после завершения работы конструктора становится пустым
`optional{ t }` `opt = t`	Копирует `t` в `optional`
`optional{ move(t) }` `opt = move(t)`	Переносит `t` в `optional`
`opt->mbr`	Разыменовывает член; предоставляет доступ к члену `mbr` объекта, содержащемуся в `opt`
`*opt` `opt.value()`	Возвращает ссылку на объект, содержащийся в `opt`; `value()` проверяет наличие пустых значений и выдает `bad_optional_access`
`opt.value_or(T{... })`	Если `opt` содержит объект, возвращает копию; иначе возвращает аргумент
`bool{ opt }` `opt.has_value()`	Возвращает `true`, если `opt` содержит объект, иначе — `false`
`opt1.swap(opt2)` `swap(opt1, opt2)`	Меняет местами объекты, содержащиеся в `opt1` и `opt2`
`opt.reset()`	Уничтожает объект, содержащийся в `opt`, который становится пустым после `reset`
`opt.emplace(...)`	Создает тип на месте, перенаправляя все аргументы в соответствующий конструктор
`make_optional(...)`	Удобная функция для построения `optional`; передает аргументы соответствующему конструктору

`opt1 == opt2`
`opt1 != opt2`
`opt1 > opt2`
`opt1 >= opt2`
`opt1 < opt2`
`opt1 <= opt2`
	При оценке равенства двух `optional`-объектов возвращает `true`, если оба они пусты или оба содержат объекты и эти объекты равны; иначе `false`. Для сравнения, пустой `optional` всегда меньше, чем `optional`, содержащий значение. В противном случае результатом является сравнение содержащихся типов.

pair

pair — это шаблон класса, который содержит два объекта разных типов в одном объекте. Объекты упорядочены, и можно получить к ним доступ через члены `first` («первый») и `second` («второй»). `pair` поддерживает операторы сравнения, имеет конструкторы копирования/переноса по умолчанию и работает с синтаксисом структурированной привязки.

`std::pair` содержится в stdlib в заголовке `<utility>`, а в Boost имеется `boost::pair` в заголовке `<boost/pair.hpp>`.

ПРИМЕЧАНИЕ

Boost также имеет `boost::ressed pair`, доступный в заголовке `<boost/compressed_pair.hpp>`. Он немного более эффективен, когда один из членов пуст.

Сначала создаются несколько простых типов, из которых можно создать пару, например классы `Socialite` и `Valet` в листинге 12.8.

Листинг 12.8. Классы Socialite и Valet

```
#include <utility>

struct Socialite { const char* birthname; };
struct Valet { const char* surname; };
Socialite bertie{ "Wilberforce" };
Valet reginald{ "Jeeves" };
```

Теперь при наличии `Socialite` и `Valet`, `bertie` и `reginald` можно создать `std::pair` и поэкспериментировать с извлечением элементов. Листинг 12.9 использует члены `first` и `second` для доступа к содержащимся типам.

Листинг 12.9. std::pair поддерживает извлечение членов

```
TEST_CASE("std::pair permits access to members") {
  std::pair<Socialite, Valet> inimitable_duo{ bertie, reginald }; ❶
  REQUIRE(inimitable_duo.first.birthname == bertie.birthname); ❷
  REQUIRE(inimitable_duo.second.surname == reginald.surname); ❸
}
```

`std::pair` создается путем передачи объектов, которые нужно скопировать ❶. Члены `std::pair` `first` и `second` используются, чтобы извлечь `Socialite` ❷ и `Valet` ❸ из `inimitable_duo`. Затем можно сравнить их члены `birthname` и `surname` с оригиналами.

В листинге 12.10 приведен синтаксис извлечения членов `std::pair` и структурированной привязки.

Здесь используется синтаксис структурированной привязки ❶ для извлечения ссылок на члены `first` и `secondinimitable_duo` в `idle_rich` и `butler`. Как и в листинге 12.9, гарантируется, что `birthname` ❷ и `surname` ❸ соответствуют оригиналам.

Листинг 12.10. std::pair поддерживает синтаксис структурированной привязки

```
TEST_CASE("std::pair works with structured binding") {
  std::pair<Socialite, Valet> inimitable_duo{ bertie, reginald };
  auto& [idle_rich, butler] = inimitable_duo; ❶
  REQUIRE(idle_rich.birthname == bertie.birthname); ❷
  REQUIRE(butler.surname == reginald.surname); ❸
}
```

Неполный список поддерживаемых операций

В таблице 12.3 приведен список большинства поддерживаемых операций `std::pair`. В этой таблице `pr` означает `std::pair <A, B>`, `a` – объект типа `A`, а `b` – объект типа `B`.

Таблица 12.3. Большинство поддерживаемых операций `std::pair`

Операция	Примечания
`pair{}`	Создает пустой `pair`
`pair{ pr }`	Создает `pair` из `pr` с использованием конструктора копирования
`pair{ move(pr) }`	Создает `pair` из `pr` с использованием конструктора переноса
`pair{ a, b }`	Создает `pair`, копируя `a` и `b`
`pair{ move(a) move(b) }`	Создает `pair`, перенося `a` и `b`
`pr1 = pr2`	Копирует присваивания из `pr2`
`pr1 = move(pr2)`	Переносит присваивания из `pr2`
`pr.first` `get<0>(pr)`	Возвращает ссылку на элемент `first`
`pr.second` `get<1>(pr)`	Возвращает ссылку на элемент `second`
`get(pr)`	Если `first` и `second` имеют разные типы, возвращает ссылку на элемент типа `T`
`pr1.swap(pr2)` `swap(pr1, pr2)`	Меняет местами объекты, содержащиеся в `pr1` и `pr2`
`make_pair(a, b)`	Меняет местами объекты, содержащиеся в `pr1` и `pr2`
`pr1 == pr2`	Равенство верно, если `first` и `second` равны
`pr1 != pr2` `pr1 > pr2` `pr1 >= pr2` `pr1 < pr2` `pr1 <= pr2`	Сравнение больше /меньше начинается с `first`. Если члены `first` равны, сравниваются члены `second`

tuple

tuple (кортеж) — это шаблон класса, который принимает произвольное количество разнородных элементов. Это обобщение `pair`, но `tuple` не выставляет свои члены как `first`, `second` и т. д. Вместо этого используется шаблон функции, не являющейся членом, `get` для извлечения элементов.

`std::tuple` и `std::get` содержатся в stdlib в заголовке `<tuple>`, а в Boost имеются `boost::tuple` и `boost::get` в заголовке `<boost/tuple/tuple.hpp>`.

Давайте добавим третий класс, `Acquaintance`, чтобы протестировать `tuple`:

```
struct Acquaintance { const char* nickname; };
Acquaintance hildebrand{ "Tuppy" };
```

Существуют два режима для извлечения этих элементов с использованием `get`. В первом случае всегда можно предоставить параметр шаблона, соответствующий нулевому индексу элемента, который нужно извлечь. В случае если `tuple` не содержит элементов с одинаковыми типами, можно альтернативно предоставить параметр шаблона, соответствующий типу элемента, который нужно извлечь, как показано в листинге 12.11.

Листинг 12.11. std::tuple поддерживает извлечение членов и синтаксис структурированной привязки.

```
TEST_CASE("std::tuple permits access to members with std::get") {
  using Trio = std::tuple<Socialite, Valet, Acquaintance>;
  Trio truculent_trio{ bertie, reginald, hildebrand };
  auto& bertie_ref = std::get<0>(truculent_trio); ❶
  REQUIRE(bertie_ref.birthname == bertie.birthname);

  auto& tuppy_ref = std::get<Acquaintance>(truculent_trio); ❷
  REQUIRE(tuppy_ref.nickname == hildebrand.nickname);
}
```

Вы можете создать `std::tuple` аналогично тому, как был создан `std::pair`. Сначала извлекается член `Socialite` с помощью `get<0>` ❶. Поскольку `Socialite` является первым параметром шаблона, используется 0 для параметра шаблона `std::get`. Затем извлекается член `Acquaintance` с помощью `std::get<Acquaintance>` ❷. Поскольку существует только один элемент типа `Acquaintance`, разрешено использовать этот режим получения доступа.

Как и `pair`, `tuple` также допускает синтаксис структурированной привязки.

Неполный список поддерживаемых операций

В таблице 12.4 приведен список большинства поддерживаемых операций `std::tuple`. В этой таблице `tp` означает `std::tuple<A, B>`, `a` — объект типа `A`, а `b` — объект типа `B`.

Таблица 12.4. Большинство поддерживаемых операций `std::tuple`

Операция	Примечания
`tuple<…>{ [alc] }`	Создает пустой `tuple`. Использует `std::allocate` как распределитель по умолчанию `alc`
`tuple<…>{ [alc], tp }`	Создает `tuple` из `tp` с использованием конструктора копирования. Использует `std::allocate` как распределитель по умолчанию `alc`
`tuple<...>{ [alc], move(tp) }`	Создает `tuple` из `tp` с использованием конструктора переноса. Использует `std::allocate` как распределитель по умолчанию `alc`
`tuple<...>{ [alc], a, b }`	Создает `tuple`, копируя `a` и `b`. Использует `std::allocate` как распределитель по умолчанию `alc`
`tuple<...>{ [alc], move(a), move(b) }`	Создает `tuple`, перенося `a` и `b`. Использует `std::allocate` как распределитель по умолчанию `alc`
`tp1 = tp2`	Копирует присваивания из `tp2`
`tp1 = move(tp2)`	Переносит присваивания из `tp2`
`get<i>(tp)`	Возвращает ссылку на `i`-й элемент (начиная с 0)
`get<T>(tp)`	Возвращает ссылку на элемент типа `T`. Выдает ошибку, если несколько элементов имеют этот тип
`tp1.swap(tp2)` `swap(tp1, tp2)`	Меняет местами объекты, содержащиеся в `tp1` и `tp2`
`make_tuple(a, b)`	Удобная функция для создания `tuple`
`tuple_cat<...>(tp1, tp2)`	Объединяет все `tuple`, переданные в качестве аргументов

`tp1 == tp2`
`tp1 != tp2`
`tp1 > tp2`
`tp1 >= tp2`
`tp1 < tp2`
`tp1 <= tp2`
	Равенство верно, если все элементы равны
	Сравнение больше /меньше вычисляется от первого до последнего элемента

any

any — это класс, который хранит отдельные значения любого типа. Это *не* шаблон класса. Чтобы преобразовать `any` в конкретный тип, используется *приведение any*, которое является шаблоном функции без членов. Любые преобразования при приведении являются безопасными для типов; если вы пытаетесь привести `any` к неподходящему типу, будет сгенерировано исключение. Используя `any`, можно выполнять некоторые виды общего программирования *без шаблонов*.

`std::any` содержится в stdlibb заголовке `<any>`, а в Boost имеется `boost::any` в заголовке `<boost/any.hpp>`.

Чтобы хранить значение в `any`, используется шаблон метода `emplace`. Он принимает один параметр шаблона, соответствующий типу, который нужно хранить в `any` (тип хранения). Все аргументы, передаваемые в `emplace`, передаются соответствующему конструктору для данного типа хранилища. Чтобы извлечь значение, используется `any_cast`, который принимает параметр шаблона, соответствующий текущему типу хранилища `any` (называемый состоянием `any`). `any` передается в качестве единственного параметра `any_cast`. Пока состояние `any` соответствует параметру шаблона, получается желаемый тип. Если состояние не совпадает, выводится исключение `bad_any_cast`.

Листинг 12.12 показывает эти основные взаимодействия с `std::any`.

Листинг 12.12. std::any и std::any_cast позволяют извлекать конкретные типы

```
#include <any>

struct EscapeCapsule {
  EscapeCapsule(int x) : weight_kg{ x } { }
  int weight_kg;
}; ❶

TEST_CASE("std::any allows us to std::any_cast into a type") {
  std::any hagunemnon; ❷
  hagunemnon.emplace<EscapeCapsule>(600); ❸
  auto capsule = std::any_cast<EscapeCapsule>(hagunemnon); ❹
  REQUIRE(capsule.weight_kg == 600);
  REQUIRE_THROWS_AS(std::any_cast<float>(hagunemnon), std::bad_any_cast); ❺
}
```

В коде объявляется класс `EscapeCapsule` ❶. В рамках теста создается пустой `std::any` с именем `hagunemnon` ❷. Далее используется `emplace` для хранения `EscapeCapsule` с `weight_kg = 600` ❸. Можно извлечь `EscapeCapsule` обратно, используя `std::any_cast` ❹, который хранится в новом `EscapeCapsule` с именем `capsule`. Наконец, показывается, что попытка вызвать `any_cast` для приведения `hagunemnon` в число с плавающей точкой приводит к исключению `std::bad_any_cast` ❺.

Неполный список поддерживаемых операций

В таблице 12.5 приведен список большинства поддерживаемых операций `std::any`. В этой таблице `ay` означает `std::any`, а `t` — объект типа `T`.

variant

variant (вариант) — это шаблон класса, в котором хранятся отдельные значения, типы которых ограничены пользовательским списком, представленным в качестве параметров шаблона. Вариант представляет собой типо-безопасный `union` (см. «Объ-

Таблица 12.5. Большинство поддерживаемых операций `std::any`

Операция	Примечания
`any{}`	Создает пустой объект `any`
`any{ ay }`	Создает `any` из `ay` с помощью конструктора копирования
`any{ move(ay) }`	Создает `any` из `ay` с помощью конструктора переноса
`any{ move(t) }`	Создает объект `any`, содержащий созданный на месте объект из `t`
`ay = t`	Уничтожает объект, содержащийся в данный момент в `ay`; копирует `t`
`ay = move(t)`	Уничтожает объект, содержащийся в данный момент в `ay`; перемещает `t`
`ay1 = ay2`	Копирует присваивания из `ay2`
`ay1 = move(ay2)`	Переносит присваивания из `ay2`
`ay.emplace<T>(...)`	Уничтожает объект, содержащийся в данный момент в `ay`, создает `T` на месте, передавая аргументы ... соответствующему конструктору
`ay.reset()`	Уничтожает объект, содержащийся в данный момент
`ay1.swap(ay2)` `swap(ay1, ay2)`	Меняет местами объекты, содержащиеся в `ay1` и `ay2`
`make_any<T>(...)`	Удобная функция для создания `any` создает `T` на месте, передавая аргументы ... соответствующему конструктору
`t = any_cast<T>(ay)`	Приводит `ay` к типу T. Выдает `std::bad_any_cast`, если тип `T` не соответствует типу содержащегося объекта

единения»). Он имеет много функциональных возможностей для типа `any`, но `variant` требует явного перечисления всех типов, которые будут храниться в нем.

`std::variant` содержится в stdlib в заголовке `<variant>`, а в Boost имеется `boost::variant` в заголовке `<boost/variant.hpp>`.

Листинг 12.13 демонстрирует создание другого типа с именем `BugblatterBeast` для `variant`, который будет содержаться вместе с `EscapeCapsule`.

Листинг 12.13. std::variable может содержать объект из одного из списков предопределенных типов

```
#include <variant>

struct BugblatterBeast {
  BugblatterBeast() : is_ravenous{ true }, weight_kg{ 20000 } { }
  bool is_ravenous;
  int weight_kg; ❶
};
```

Помимо того что он содержит элемент `weight_kg` ❶, `BugblatterBeast` полностью независим от `EscapeCapsule`.

Создание variant

`variant` может быть создан по умолчанию, только если выполняется одно из двух условий:

- первый параметр шаблона является конструируемым по умолчанию;
- он принадлежит типу `monostate`, предназначенному для сообщения о том, что `variant` может иметь пустое состояние.

Поскольку `BugblatterBeast` является конструируемым по умолчанию (то есть имеет конструктор по умолчанию), сделайте его первым типом в списке параметров шаблона, чтобы вариант также был конструируемым по умолчанию, например:

```
std::variant<BugblatterBeast, EscapeCapsule> hagunemnon;
```

Чтобы хранить значение в `variant`, используется шаблон метода `emplace`. Как и в случае с `any`, `variant` принимает один параметр шаблона, соответствующий типу, который нужно хранить. Этот параметр шаблона должен содержаться в списке параметров шаблона для `variant`. Чтобы извлечь значение, используется один из шаблонов функций, не являющихся членами, `get` или `get_if`. Они принимают либо нужный тип, либо индекс в списке параметров шаблона, соответствующего требуемому типу. Если `get` терпит неудачу, он генерирует исключение `bad_variant_access`, а `get_if` возвращает `nullptr`.

Можно определить, какой тип соответствует текущему состоянию варианта, используя член `index()`, который возвращает индекс типа текущего объекта в списке параметров шаблона.

В листинге 12.14 показано, как использовать `emplace` для изменения состояния `variant` и `index` для определения типа содержащегося объекта.

Листинг 12.14. std::get позволяет извлекать конкретные типы из std::option

```
TEST_CASE("std::variant") {
  std::variant<BugblatterBeast, EscapeCapsule> hagunemnon;
  REQUIRE(hagunemnon.index() == 0); ❶

  hagunemnon.emplace<EscapeCapsule>(600); ❷
  REQUIRE(hagunemnon.index() == 1); ❸

  REQUIRE(std::get<EscapeCapsule>(hagunemnon).weight_kg == 600); ❹
  REQUIRE(std::get<1>(hagunemnon).weight_kg == 600); ❺
  REQUIRE_THROWS_AS(std::get<0>(hagunemnon), std::bad_variant_access); ❻
}
```

После построения по умолчанию `hagunemnon` вызов `index` возвращает 0, потому что это индекс правильного параметра шаблона ❶. Затем устанавливается `EscapeCapsule` ❷, который заставляет `index` возвращать 1 вместо этого ❸.

И `std::get<EscapeCapsule>` ❹, и `std::get<1>` ❺ показывают идентичные способы извлечения содержимого типа. Наконец, попытка вызвать `std::get` для получения типа, который не соответствует текущему состоянию `variant`, приводит к значению `bad_variant_access` ❻.

Можно использовать функцию `std::visit`, не являющуюся членом, чтобы применить вызываемый объект к `variant`. Это имеет преимущество в том, что отправляется правильная функция для обработки того, что содержится в объекте, без необходимости явно указывать ее с помощью `std::get`. Листинг 12.15 показывает основное использование.

Листинг 12.15. std::visit позволяет добавить вызываемый объект к содержащимся типам std::variant.

```
TEST_CASE("std::variant") {
  std::variant<BugblatterBeast, EscapeCapsule> hagunemnon;
  hagunemnon.emplace<EscapeCapsule>(600); ❶
  auto lbs = std::visit([](auto& x) { return 2.2*x.weight_kg; }, hagunemnon); ❷
  REQUIRE(lbs == 1320); ❸
}
```

Сначала вызывается `emplace` для сохранения значения 600 в `hagunemnon` ❶. Поскольку и `BugblatterBeast`, и `EscapeCapsule` имеют член `weight_kg`, можно использовать `std::visit` на `hagunemnon` с лямбда-выражением, которое выполняет правильное преобразование (2, 2 фунта на кг) в поле `weight_kg` ❷ и возвращает результат ❸ (обратите внимание, что не нужно включать информацию о любом из типов).

Сравнение variant и any

Вселенная достаточно велика, чтобы вместить как `any`, так и `variant`. Невозможно рекомендовать что-то одно вообще, потому что у каждого из них есть свои сильные и слабые стороны.

`any` более гибкий; он может принимать *любой* тип, тогда как `variant` может содержать только объект заранее определенного типа. Он также в основном избегает шаблонов, поэтому с ним проще программировать.

`variant` менее гибок, что делает его более безопасным. Используя функцию `visit`, вы можете проверить безопасность операций во время компиляции. С `any` нужно было бы создавать собственную функциональность, похожую на `visit`, и для этого потребуется проверка во время выполнения (например, результата `any_cast`).

Наконец, `variant` может быть более производительным, чем `any`. Зато `any` разрешено выполнять динамическое размещение, если содержащийся тип слишком велик, а `variant` — нет.

Неполный список поддерживаемых операций

В таблице 12.6 приведен список большинства поддерживаемых операций `std::option`. В этой таблице `vt` означает `std::variant`, а `t` — объект типа `T`.

Таблица 12.6. Большинство поддерживаемых операций `std::option`

Операция	Примечания
`variant<...>{}`	Создает пустой объект `variant`. Первый параметр шаблона должен быть конструируемым по умолчанию
`variant<...>{ vt }`	Создает `variant` из `vt` с использованием конструктора копирования
`variant<...>{ move(vt) }`	Создает `variant` из `vt` с использованием конструктора переноса
`variant<...>{ move(t) }`	Создает объект `variant`, содержащий созданный на месте объект
`vt = t`	Уничтожает объект, содержащийся в `vt`; копирует `t`
`vt = move(t)`	Уничтожает объект, содержащийся в `vt`; переносит `t`
`vt1 = vt2`	Копирует присваивания из `vt2`
`vt1 = move(vt2)`	Переносит присваивания из `vt2`
`vt.emplace<T>(...)`	Уничтожает объект, который в данный момент содержится в `vt`; создает `T` на месте, передавая аргументы ... соответствующему конструктору
`vt.reset()`	Уничтожает текущий содержащийся объект
`vt.index()`	Возвращает нулевой индекс типа текущего содержащегося объекта. Порядок определяется параметрами шаблона `std::variant`
`vt1.swap(vt2)` `swap(vt1, vt2)`	Меняет местами объекты, содержащиеся в `vt1` и `vt2`
`make_variant<T>(...)`	Удобная функция для построения `variant`; создает `T` на месте, передавая аргументы... соответствующему конструктору
`std::visit(callable, vt)`	Вызывает `callable` с содержащимся объектом
`std::holds_alternative<T>(vt)`	Возвращает `true`, если тип содержащегося объекта — `T`
`std::get<i>(vt)` `std::get<T>(vt)`	Возвращает содержащийся объект, если его тип `T` или `i`-й тип. В противном случае выдает исключение `std::bad_variant_access`
`std::get_if<i>(&vt)` `std::get_if<T>(&vt)`	Возвращает указатель на содержащийся объект, если его тип `T` или `i`-й тип. В противном случае возвращает `nullptr`

`vt1 == vt2`
`vt1 != vt2`
`vt1 > vt2`
`vt1 >= vt2`
`vt1 < vt2`
`vt1 <= vt2`
	Сравнивает содержащиеся объекты `vt1` и `vt2`

Дата и время

В stdlib и Boost доступно несколько библиотек, которые обрабатывают даты и время. При обработке календарных дат и времени обратите внимание на библиотеку DateTime в Boost. Чтобы узнать текущее время или измерить прошедшее время, загляните в библиотеки Chrono в Boost или stdlib и в библиотеку Timer в Boost.

DateTime в Boost

Библиотека DateTime в Boost поддерживает программирование дат с помощью системы, основанной на григорианском календаре, наиболее широко используемом гражданским календарем в мире. Календари сложнее, чем они могут показаться на первый взгляд. Например, рассмотрим следующий отрывок из введения в календари (Introduction to Calendars) Военно-морской обсерватории США, в котором описываются високосные года.

> Каждый год, который кратен четырем, является високосным, за исключением тех лет, которые кратны 100, но эти годы являются високосными, если их номер кратен 400. Например, 1700, 1800 и 1900 годы не високосные, но 2000 год — високосный.

Чтобы не создавать свои собственные функции солнечного календаря, просто добавьте средства программирования даты `DateTime` со следующим заголовком:

```
#include <boost/date_time/gregorian/gregorian.hpp>
```

Основным используемым типом является `boost::gregorian::date` — основной интерфейс для программирования даты.

Создание даты

Для создания даты доступно несколько вариантов. Можно создать дату по умолчанию, которая устанавливает специальное значение `boost::gregorian::not_a _date_time`. Чтобы построить `date` с допустимой датой, можно использовать конструктор, который принимает три аргумента: год, месяц и дату. Следующее утверждение создает `date d` с датой 15 сентября 1986 года:

```
boost::gregorian::date d{ 1986, 9, 15 };
```

Кроме того, можно собрать дату из строки, используя служебную функцию `boost::gregorian::from_string`, например, так:

```
auto d = boost::gregorian::from_string("1986/9/15");
```

Если передать недопустимую дату, конструктор `date` выдаст исключение, например `bad_year`, `bad_day_of_month` или `bad_month`. Например, в листинге 12.16 делается попытка создать дату со значением 32 сентября 1986 года.

Листинг 12.16. Конструктор boost::gregorian::date генерирует исключения для неправильных дат

```
TEST_CASE("Invalid boost::Gregorian::dates throw exceptions") {
  using boost::gregorian::date;
  using boost::gregorian::bad_day_of_month;

  REQUIRE_THROWS_AS(date(1986, 9, 32), bad_day_of_month); ❶
}
```

Поскольку 32 сентября не является допустимым днем месяца, конструктор даты генерирует исключение `bad_day_of_month` ❶.

ПРИМЕЧАНИЕ

Из-за ограничения в Catch нельзя использовать фигурную инициализацию для `date` в макросе `REQUIRE_THROWS_AS` ❶.

Можно получить текущий день из среды, используя функцию, не являющуюся членом, `boost::gregorian::day_clock::local_day` или `boost::gregorian::day_clock::universal_day`, чтобы получить местный день на основе настроек часового пояса системы и дня UTC соответственно:

```
auto d_local = boost::gregorian::day_clock::local_day();
auto d_univ = boost::gregorian::day_clock::universal_day();
```

После создания даты нельзя изменить ее значение (она *неизменна*). Тем не менее даты поддерживают создание и присваивание копии.

Доступ к членам даты

Вы можете проверить особенности `date` с помощью многих методов `const`. В таблице 12.7 приведен их неполный список. В этой таблице `d` означает `date`.

Таблица 12.7. Наиболее поддерживаемые методы доступа `boost::gregorian::date`

Метод доступа	Примечания
`d.year()`	Возвращает годовую часть даты
`d.month()`	Возвращает месячную часть даты
`d.day()`	Возвращает дневную часть даты
`d.day_of_week()`	Возвращает день недели в виде `enum` типа `greg_day_of_week`
`d.day_of_year()`	Возвращает день года (от 1 до 366 включительно)
`d.end_of_month()`	Возвращает объект даты, равный последнему дню месяца `d`
`d.is_not_a_date()`	Возвращает `true`, если `d` — не дата
`d.week_number()`	Возвращает номер недели в системе ISO 8601

В листинге 12.17 показано, как построить дату и использовать методы доступа из таблицы 12.7.

Листинг 12.17. boost::gregorian::date поддерживает основные функции календаря

```
TEST_CASE("boost::gregorian::date supports basic calendar functions") {
  boost::gregorian::date d{ 1986, 9, 15 }; ❶
  REQUIRE(d.year() == 1986); ❷
  REQUIRE(d.month() == 9); ❸
  REQUIRE(d.day() == 15); ❹
  REQUIRE(d.day_of_year() == 258); ❺
  REQUIRE(d.day_of_week() == boost::date_time::Monday); ❻
}
```

Здесь создается `date` со значением 15 сентября 1986 г. ❶. Оттуда извлекается год ❷, месяц ❸, день ❹, день года ❺ и день недели ❻.

Математические расчеты

Можно выполнить простые математические вычисления для дат. При вычитании одной даты из другой получается `boost::gregorian::date_duration`. Основная функция `date_duration` — хранение целого числа дней, которое можно извлечь с помощью метода `days`. В листинге 12.18 показано, как вычислить количество дней, прошедших между двумя объектами даты.

Листинг 12.18. Вычитание объектов boost::gregorian::date дает boost::gregorian::date_duration.

```
TEST_CASE("boost::gregorian::date supports calendar arithmetic") {
  boost::gregorian::date d1{ 1986, 9, 15 }; ❶
  boost::gregorian::date d2{ 2019, 8, 1 }; ❷
  auto duration = d2 - d1; ❸
  REQUIRE(duration.days() == 12008); ❹
}
```

Здесь создаются `date` со значениями 15 сентября 1986 г. ❶ и 1 августа 2019 г. ❷. Эти две даты вычитаются, чтобы получить `date_duration` ❸. Используя метод `days`, можно извлечь количество дней между двумя датами ❹.

Также можно создать `date_duration`, используя аргумент типа `long`, соответствующий количеству дней. Можно добавить `date_duration` к дате, чтобы получить другую, как показано в листинге 12.19.

Листинг 12.19. Добавление date_duration к дате приводит к созданию другой даты

```
TEST_CASE("date and date_duration support addition") {
  boost::gregorian::date d1{ 1986, 9, 15 }; ❶
  boost::gregorian::date_duration dur{ 12008 }; ❷
  auto d2 = d1 + dur; ❸
  REQUIRE(d2 == boost::gregorian::from_string("2019/8/1")); ❹
}
```

Здесь создается date со значением 15 сентября 1986 г. ❶ и duration со значением 12 008 дней ❷. Из листинга 12.18 вы знаете, что этот день плюс 12008 дает 1 августа 2019 года. Итак, после сложения ❸ получившийся день будет таким же, как ожидается ❹.

Период дат

Период дат представляет собой интервал между двумя датами. DateTime предоставляет класс boost::gregorian::date_period, который имеет три конструктора, как описано в таблице 12.8. В этой таблице конструкторы d1 и d2 означают аргументы date, а dp — date_period.

Таблица 12.8. Поддерживаемые конструкторы boost::gregorian::date_period

Метод доступа	Примечания
date_period{ d1, d2 }	Создает период, включающий d1, но не d2; если d2 <= d1, выдается ошибка
date_period{ d, n_days }	Создает период от d до d+n_days
date_period{ dp }	Конструктор копирования

Класс date_period поддерживает множество операций, таких как метод contain, который принимает аргумент date и возвращает значение true, если аргумент содержится в period. В листинге 12.20 продемонстрирована эта операция.

Листинг 12.20. Использование метода contains для boost::gregorian::date_period для определения того, попадает ли дата в определенный интервал времени

```
TEST_CASE("boost::gregorian::date supports periods") {
  boost::gregorian::date d1{ 1986, 9, 15 }; ❶
  boost::gregorian::date d2{ 2019, 8, 1 }; ❷
  boost::gregorian::date_period p{ d1, d2 }; ❸
  REQUIRE(p.contains(boost::gregorian::date{ 1987, 10, 27 })); ❹
}
```

Здесь создаются две даты, 15 сентября 1986 г. ❶ и 1 августа 2019 г. ❷, которые используются для построения date_period ❸. Используя метод contains, можно определить, что date_period содержит дату 27 октября 1987 г. ❹.

Таблица 12.9 содержит неполный список других операций date_period. В этой таблице p, p1 и p2 — классы date_period, а d – date.

Другие функции DateTime

Библиотека Boost DateTime содержит три широких категории программирования:

- **Дата.** Программирование даты — это программирование на основе календаря, и это мы только что рассмотрели.

Таблица 12.9. Поддерживаемые операции `boost::gregorian::date_period`

Метод доступа	Примечания
`p.begin()`	Возвращает первый день
`p.last()`	Возвращает последний день
`p.length()`	Возвращает количество содержащихся дней
`p.is_null()`	Возвращает `true`, если период некорректен (например, дата конца меньше даты начала)
`p.contains(d)`	Возвращает `true`, если `d` входит в `p`
`p1.contains(p2)`	Возвращает `true`, если все даты периода `p2` входят в `p1`
`p1.intersects(p2)`	Возвращает `true`, если любая дата периода `p2` входит в `p1`
`p.is_after(d)`	Возвращает `true`, если `p` находится за `d`.
`p.is_before(d)`	Возвращает `true`, если `p` находится перед `d`

- **Время.** Программирование времени, которое позволяет работать с часами с микросекундным разрешением, доступно в заголовке `<boost/date_time/posix_time/posix_time.hpp>`. Механика похожа на программирование дат, но вы работаете с часами вместо григорианских календарей.
- **Местное время.** Программирование местного времени — это просто программирование времени с учетом часового пояса. Он доступен в заголовке `<boost/date_time/time_zone_base.hpp>`.

ПРИМЕЧАНИЕ

В целях экономии мы не будет подробно рассматривать время и программирование по местному времени. См. документацию Boost для получения информации и примеров.

Chrono

Библиотека stdlib Chrono предоставляет различные часы в заголовке `<chrono>`. Обычно они используются, когда нужно запрограммировать что-то, что зависит от времени, или для выставления таймеров в коде.

ПРИМЕЧАНИЕ

Boost также предлагает библиотеку Chrono в заголовке `<boost/chrono.hpp>`. Это расширенный набор библиотеки Chrono stdlib, который включает, например, специфичные для процесса и потока часы и пользовательские форматы вывода времени.

Часы

Три типа часов доступны в библиотеке Chrono; каждый из них предоставляет отдельную гарантию, и все они находятся в пространстве имен `std::chrono`:

- `std::chrono::system_clock` — это системные часы реального времени. Иногда их также называют настенными часами, идущими в режиме реального времени с даты начала реализации. Большинство реализаций определяют дату начала Unix как полночь 1 января 1970 года;
- `std::chrono::stable_clock` гарантирует, что его значение никогда не уменьшится. Это может показаться абсурдным, но измерение времени сложнее, чем кажется. Например, система может столкнуться с дополнительными секундами или неточными часами;
- у `std::chrono::high_resolution_clock` самый короткий доступный период такта: такт — это наименьшее атомное изменение, которое могут измерить часы.

Каждый из этих трех часов поддерживает статическую функцию-член `now`, которая возвращает момент времени, соответствующий текущему значению часов.

Моменты времени

Chrono кодирует *моменты времени*, используя тип `std::chrono::time_point`. С точки зрения пользователя объекты `time_point` очень просты. Они предоставляют метод `time_since_epoch`, который возвращает количество времени, прошедшего между моментом времени и *эпохой* часов. Это прошедшее время называется *длительностью*.

Эпоха — это эталонный момент времени, определяемый реализацией, обозначающий начало часов. Эпоха Unix (или время POSIX) начинается с 1 января 1970 года, тогда как Эпоха Windows начинается 1 января 1601 года (соответствует началу 400-летнего цикла григорианского календаря).

Метод `time_since_epoch` — не единственный способ получить длительность от точки `time_since`. Можно получить длительность между двумя объектами `time_point`, вычитая их.

Длительность

`std::chrono::duration` представляет время между двумя объектами `time_point`. Длительность предоставляет метод `count`, который возвращает количество тактов в длительности.

В листинге 12.21 показано, как получить текущее время от каждого из трех доступных часов, извлечь время, прошедшее с эпохи каждого из часов как длительность, а затем преобразовать их в такты.

Текущее время получается из `system_clock` ❶, `high_resolution_clock` ❷ и `stable_clock` ❸. Для каждых часов момент времени конвертируется в длительность, начиная с эпохи часов, используя метод `time_since_epoch`. Немедленно вызывается `count`

для полученной длительности, чтобы получить счетчик тактов, который должен быть больше нуля ❹❺❻.

Листинг 12.21. std::chrono поддерживает несколько видов часов

```
TEST_CASE("std::chrono supports several clocks") {
  auto sys_now = std::chrono::system_clock::now(); ❶
  auto hires_now = std::chrono::high_resolution_clock::now(); ❷
  auto steady_now = std::chrono::steady_clock::now(); ❸

  REQUIRE(sys_now.time_since_epoch().count() > 0); ❹
  REQUIRE(hires_now.time_since_epoch().count() > 0);❺
  REQUIRE(steady_now.time_since_epoch().count() > 0); ❻
}
```

В дополнение к получению длительностей из временных точек можно создавать их напрямую. Пространство имен `std::chrono` содержит вспомогательные функции для генерации длительностей. Для удобства Chrono предлагает несколько пользовательских литералов длительности в пространстве имен `std::literals::chrono_literals`. Они предоставляют некоторый синтаксический сахар, удобный синтаксис языка, который облегчает жизнь разработчику, для определения литералов длительности.

В таблице 12.10 показаны вспомогательные функции и их буквальные эквиваленты, где каждое выражение соответствует продолжительности часа.

Таблица 12.10. Вспомогательные функции `std::chrono` и пользовательские литералы для создания длительностей

Вспомогательная функция	Литеральный эквивалент
`nanoseconds(3600000000000)`	`3600000000000ns`
`microseconds(3600000000)`	`3600000000us`
`milliseconds(3600000)`	`3600000ms`
`seconds(3600)`	`3600s`
`minutes(60)`	`60m`
`hours(1)`	`1h`

Например, в листинге 12.22 показано, как создать длительность в 1 секунду с помощью `std::chrono::seconds`, а другую — в 1000 миллисекунд, используя литерал длительности `ms`.

Листинг 12.22. std::chrono поддерживает множество сравнимых единиц измерения

```
#include <chrono>
TEST_CASE("std::chrono supports several units of measurement") {
  using namespace std::literals::chrono_literals; ❶
  auto one_s = std::chrono::seconds(1); ❷
  auto thousand_ms = 1000ms; ❸
  REQUIRE(one_s == thousand_ms); ❹
}
```

Здесь вводится пространство имен `std::literals::chrono_literals`, чтобы иметь доступ к литералам длительности ❶. Создается одна длительность с именем `one_s` из вспомогательной функции `seconds` ❷, а другая — с именем `thousand_ms` из литерала длительности `ms` ❸. Они эквивалентны, потому что секунда содержит тысячу миллисекунд ❹.

Chrono предоставляет шаблон функции `std::chrono::duration_cast` для приведения длительности из одного блока в другой. Как и в других шаблонах функций, связанных с приведением типов, таких как `static_cast`, `duration_cast` принимает один параметр шаблона, соответствующий целевой продолжительности, и один аргумент, соответствующий продолжительности, к которой необходимо привести.

В листинге 12.23 показано, как преобразовать длительность `nanosecond` в `second`.

Листинг12.23. std::chrono поддерживает std::chrono::duration_cast.

```
TEST_CASE("std::chrono supports duration_cast") {
  using namespace std::chrono; ❶
  auto billion_ns_as_s = duration_cast<seconds❷>(1'000'000'000ns❸);
  REQUIRE(billion_ns_as_s.count() == 1); ❹
}
```

Во-первых, вводится пространство имен `std::chrono` для быстрого доступа к `duration_cast`, вспомогательным функциям длительности и литералам длительности ❶. Затем используется литерал длительности `ns`, чтобы указать длительность ❸ в миллиардсекундах, которая передается в качестве аргумента в `duration_cast`. Параметр шаблона `duration_cast` задается в секундах ❷, поэтому итоговая длительность `billion_ns_as_s` равна 1 секунде ❹.

Ожидание

Иногда вы будете использовать длительность, чтобы указать период времени, в течение которого программа будет ждать. stdlib предоставляет примитивы параллелизма в заголовке `<thread>`, который содержит функцию, не являющуюся членом, `std::this_thread::sleep_for`. Функция `sleep_for` принимает аргумент длительности, соответствующий времени, которое долго необходимо, чтобы текущий поток выполнения ожидал или «спал».

В листинге 12.24 показано, как использовать `sleep_for`.

Листинг 12.24. std::chrono работает с <thread>, чтобы перевести текущий поток в спящий режим

```
#include <thread>
#include <chrono>

TEST_CASE("std::chrono used to sleep") {
  using namespace std::literals::chrono_literals; ❶
  auto start = std::chrono::system_clock::now(); ❷
  std::this_thread::sleep_for(100ms); ❸
  auto end = std::chrono::system_clock::now(); ❹
  REQUIRE(end - start >= 100ms); ❺
}
```

Как и прежде, вводится пространство имен `chrono_literals`, чтобы иметь доступ к литералам длительности ❶. Текущее время записывается в соответствии с `system_clock`, сохраняя результирующую точку времени в переменной `start` ❷. Затем вызывается `sleep_for` с длительностью 100 миллисекунд (десятой доли секунды) ❸. Затем снова записывается текущее время, сохраняя полученную точку времени в `end` ❹. Поскольку программа спала в течение 100 миллисекунд между вызовами `std::chrono::system_clock`, длительность, полученная в результате вычитания `start` из `end`, должна составлять не менее 100 миллисекунд ❺.

Отсчет времени

Чтобы оптимизировать код, необходимы точные измерения. Можно использовать Chrono, чтобы измерить, сколько времени занимает серия операций. Это позволяет установить, что конкретный путь к коду действительно отвечает за наблюдаемые проблемы с производительностью. Это также позволяет установить объективную меру для оценки прогресса усилий по оптимизации.

Библиотека BoostTimer содержит класс `boost::timer::auto_cpu_timer` в заголовке `<boost/timer/timer.hpp>`, который является объектом RAII, начинает синхронизацию в своем конструкторе и останавливает синхронизацию в деструкторе.

Можно создать свой собственный временный класс `Stopwatch` («Секундомер»), используя только библиотеку Chrono в stdlib. Класс `Stopwatch` может хранить ссылку на объект `duration`. В деструкторе `Stopwatch` можно установить `duration` по его ссылке. В листинге 12.25 приведена реализация.

Листинг 12.25. Простой класс Stopwatch, который вычисляет продолжительность его жизни

```
#include <chrono>

struct Stopwatch {
  Stopwatch(std::chrono::nanoseconds& result❶)
    : result{ result }, ❷
    start{ std::chrono::high_resolution_clock::now() } { } ❸
  ~Stopwatch() {
    result = std::chrono::high_resolution_clock::now() - start; ❹
  }
private:
  std::chrono::nanoseconds& result;
  const std::chrono::time_point<std::chrono::high_resolution_clock> start;
};
```

Для `Stopwatch` секундомера требуется одна ссылка на `nanoseconds` ❶, которая сохраняется в поле результата с помощью инициализатора элемента ❷. Также сохраняется текущее время `high_resolution_clock`, устанавливая в поле `start` результат `now()` ❸. В деструкторе секундомера снова вызывается `now()` для `high_resolution_clock` и вычитается `start`, чтобы получить длительность жизни `Stopwatch`. Ссылка на результат используется, чтобы записать `duration` ❹.

В листинге 12.26 показан секундомер в действии, выполняющий миллион операций деления с плавающей точкой внутри цикла и вычисляющий среднее время, затраченное на одну итерацию.

Листинг 12.26. Использование Stopwatch для оценки времени, необходимого для деления double

```
#include <cstdio>
#include <cstdint>
#include <chrono>

struct Stopwatch {
--пропуск--
};

int main() {
  const size_t n = 1'000'000; ❶
  std::chrono::nanoseconds elapsed; ❷
  {
    Stopwatch stopwatch{ elapsed }; ❸
    volatile double result{ 1.23e45 }; ❹
    for (double i = 1; i < n; i++) {
      result /= i; ❺
    }
  }
  auto time_per_division = elapsed.count() / double{ n }; ❻
  printf("Took %gns per division.", time_per_division); { ❼
}
--------------------------------------------------------------------------------
Took 6.49622ns per division. ❼
```

Сначала инициализируется переменная *n*, равная миллиону, в которой хранится общее число итераций, которые выполнит программа ❶. Объявляется переменная `elapsed`, которая будет хранить время, прошедшее через все итерации ❷. Внутри блока объявляется `Stopwatch` и передается ссылка на `elapsed` конструктору ❸. Далее объявляется `double` под названием `result` ❹. Эта переменная помечается как `volatile`, чтобы компилятор не пытался оптимизировать цикл. Внутри цикла делается произвольное деление чисел с плавающей точкой ❺.

Как только блок завершается, `stopwatch` уничтожается. Это записывает длительность `stopwatch` в `elapsed`, которая используется для вычисления среднего числа наносекунд на итерацию цикла и сохранения в переменной `time_per_division` ❻. Программа завершается, и выводится `time_per_division` с помощью `printf` ❼.

Числовые данные

В этом разделе обсуждается обработка чисел с акцентом на общие математические функции и константы; обработка комплексных чисел; генерирование случайных чисел, числовых пределов и преобразований; вычислительные отношения.

Числовые функции

Библиотеки stdlib Numerics и Boost Math предоставляют множество числовых/математических функций. Для краткости в этой главе представлены только краткие ссылки. Подробнее об этом см. в [numerics] в стандарте ISO C++ 17 и в документации по Boost Math.

В таблице 12.11 приведен неполный список многих распространенных математических функций, не являющихся членами, доступных в библиотеке Math в stdlib.

Таблица 12.11. Неполный список общих математических функций в stdlib

Функция	Вычисляет...	Целые числа	Числа с плавающей точкой	Заголовок
`abs(x)`	Абсолютное значение x	✓		`<cstdlib>`
`div(x, y)`	Частное и остаток от деления x на y	✓		`<cstdlib>`
`abs(x)`	Абсолютное значение x		✓	`<cmath>`
`fmod(x, y)`	Остаток от деления числа с плавающей точкой x на y		✓	`<cmath>`
`remainder(x, y)`	Остаток от деления x на y со знаком	✓	✓	`<cmath>`
`fma(x, y, z)`	Умножьте первые два аргумента и добавьте их произведение к третьему аргументу; также называется сложным умножением сложения; то есть `x * y + z`	✓	✓	`<cmath>`
`max(x, y)`	Максимальное значение среди x и y	✓	✓	`<algorithm>`
`min(x, y)`	Минимальное значение среди x и y	✓	✓	`<algorithm>`
`exp(x)`	Значение e^x	✓	✓	`<cmath>`
`exp2(x)`	Значение 2^x	✓	✓	`<cmath>`
`log(x)`	Натуральный логарифм x, то есть lnx	✓	✓	`<cmath>`
`log10(x)`	Десятичный логарифм x, то есть `log10 x`	✓	✓	`<cmath>`
`log2(x)`	Базовый логарифм x, то есть `log2 x`	✓	✓	`<cmath>`
`gcd(x, y)`	Наибольшее общее кратное x и y	✓		`<numeric>`
`lcm(x, y)`	Наименьшее общее кратное x и y	✓		`<numeric>`
`erf(x)`	Функция ошибки Гаусса от x	✓	✓	`<cmath>`
`pow(x, y)`	Значение x^y	✓	✓	`<cmath>`
`sqrt(x)`	Квадратный корень из x	✓	✓	`<cmath>`

Продолжение ⇩

Таблица 12.11 *(продолжение)*

Функция	Вычисляет...	Целые числа	Числа с плавающей точкой	Заголовок
`cbrt(x)`	Кубический корень из x	✓	✓	`<cmath>`
`hypot(x, y)`	Квадратный корень из $x^2 + y^2$	✓	✓	`<cmath>`

`sin(x)`
`cos(x)`
`tan(x)`
`asin(x)`
`acos(x)`
`atan(x)`
	Значение соответствующей тригонометрической функции	✓	✓	`<cmath>`

`sinh(x)`
`cosh(x)`
`tanh(x)`
`asinh(x)`
`acosh(x)`
`atanh(x)`
	Значение соответствующей гиперболической функции	✓	✓	`<cmath>`

`ceil(x)`	Ближайшее целое число, большее или равное x	✓	✓	`<cmath>`
`floor(x)`	Ближайшее целое число, меньшее или равное x	✓	✓	`<cmath>`
`round(x)`	Ближайшее целое число, равное x; округляет от нуля при равноудаленности	✓	✓	`<cmath>`
`isfinite(x)`	Значение `true`, если x — конечное число	✓	✓	`<cmath>`
`isinf(x)`	Значение `true`, если x — бесконечное число			`<cmath>`

ПРИМЕЧАНИЕ

Другие специализированные математические функции находятся в заголовке `<cmath>`. Например, функции для вычисления полиномов Лагерра и Эрмита, эллиптические интегралы, цилиндрические функции Бесселя и Неймана и дзета-функция Римана содержатся в данном заголовке.

Комплексные числа

Комплексное число имеет вид `a+bi`, где `i` — *мнимое число*, которое при умножении на себя равно минус единице; то есть `i*i=-1`. Мнимые числа имеют применения

в теории управления, гидродинамике, электротехнике, анализе сигналов, теории чисел и квантовой физике, а также в других областях. Часть `a` комплексного числа называется его *действительной частью*, а `b` –*мнимой*.

stdlib предлагает шаблон класса `std::complex` в заголовке `<complex>`.

Он принимает параметр шаблона для базового типа реального и мнимого компонента. Этот параметр шаблона должен быть одним из основных типов с плавающей точкой.

Чтобы построить `complex`, можно передать два аргумента: действительный и мнимый компоненты. Класс `complex` также поддерживает конструктор копирования и присваивание копии.

Функции, не являющиеся членами `std::real` и `std::imag`, могут извлекать действительные и мнимые компоненты из `complex` соответственно, как показано в листинге 12.27.

Листинг 12.27. Создание std::complex и извлечение его компонентов

```
#include <complex>

TEST_CASE("std::complex has a real and imaginary component") {
  std::complex<double> a{0.5, 14.13}; ❶
  REQUIRE(std::real(a) == Approx(0.5)); ❷
  REQUIRE(std::imag(a) == Approx(14.13)); ❸
}
```

`std::complex` создается с вещественным компонентом 0.5 и мнимым компонентом 14.13 ❶. `std::real` используется для извлечения реального компонента ❷, а `std::imag` — для извлечения мнимого компонента ❸.

Таблица 12.12 содержит неполный список поддерживаемых операций `std::complex`.

Таблица 12.12. Неполный список операций `std::complex`

Операция	Примечания

`c1+c2`
`c1-c2`
`c1*c2`
`c1/c2`
	Выполняет сложение, вычитание, умножение и деление

`c+s`
`c-s`
`c*s`
`c/s`
	Преобразует скаляр `s` в сложное число с вещественным компонентом, равным скалярному значению, и мнимым компонентом, равным нулю. Это преобразование поддерживает соответствующую сложную операцию (сложение, вычитание, умножение или деление) в предыдущей строке

`real(c)`	Извлекает действительный компонент
`imag(c)`	Извлекает мнимый компонент
`abs(c)`	Вычисляет величину

Продолжение ⇩

Таблица 12.12 *(продолжение)*

Операция	Примечания
`arg(c)`	Вычисляет фазовый угол
`norm(c)`	Вычисляет квадратную величину
`conj(c)`	Вычисляет комплексно-сопряженную величину
`proj(c)`	Вычисляет сферу Римана
`sin(c)`	Вычисляет синус
`cos(c)`	Вычисляет косинус
`tan(c)`	Вычисляет тангенс
`asin(c)`	Вычисляет арксинус
`acos(c)`	Вычисляет арккосинус
`atan(c)`	Вычисляет арктангенс
`c = polar(m, a)`	Вычисляет сложное число, определяемое величиной `m` и углом `a`

Математические постоянные

Boost предлагает набор часто используемых математических постоянных в заголовке `<boost/math/constants/constants.hpp>`. Доступно более 70 констант, и можно получить их в формате `float`, `double` или `long double`, извлекая соответствующую глобальную переменную из `boost::math::float_constants`, `boost::math::double_constants` и `boost::math::long_double_constants` соответственно.

Одна из многих доступных постоянных – `four_thirds_pi`, которая приблизительно равна 4/3. Формула для вычисления объема сферы радиуса r равна $4\pi r^3/3$, так что можно использовать эту константу, чтобы упростить вычисление такого объема. В листинге 12.28 показано, как вычислить объем сферы с радиусом 10.

Листинг 12.28. Пространство имен boost::math предлагает константы

```
#include <cmath>
#include <boost/math/constants/constants.hpp>

TEST_CASE("boost::math offers constants") {
  using namespace boost::math::double_constants; ❶
  auto sphere_volume = four_thirds_pi * std::pow(10, 3); ❷
  REQUIRE(sphere_volume == Approx(4188.7902047));
}
```

Здесь используется пространство имен `boost::math::double_constants`, которое выводит все `double`-версии постоянных Boost Math ❶. Затем рассчитывается `sphere_volume`, вычисляя `four_thirds_pi` 10^3 раза ❷.

В таблице 12.13 приведены некоторые наиболее часто используемые постоянные в Boost Math.

Таблица 12.13. Некоторые из наиболее распространенных постоянных в Boost Math

Постоянная	Значение	Приблизительное вычисление	Примечание
half	1/2	0.5	
third	1/3	0.333333	
two_thirds	2/3	0.66667	
three_quarters	3/4	0.75	
root_two	√2	1.41421	
root_three	√3	1.73205	
half_root_two	√2/2	0.707106	
ln_two	ln(2)	0.693147	
ln_ten	ln(10)	2.30258	
pi	π	3.14159	Постоянная Архимеда
two_pi	2π	6.28318	Окружность круга
four_thirds_pi	4π/3	4.18879	Объем сферы
one_div_two_pi	1/(2π)	1.59155	Гауссовы интегралы
root_pi	√π	1.77245	
e	e	2.71828	Постоянная Эйлера
e_pow_pi	e^{π}	23.14069	Постоянная Гельфонда
root_e	√e	1.64872	
log10_e	log10(e)	0.434294	
degree	π/180	0.017453	Количество радиан на градус
radian	180/π	57.2957	Количество градусов на радиан
sin_one	sin(1)	0.84147	
cos_one	cos(1)	0.5403	
phi	(1 + √5)/2	1.61803	Золотое сечение Фидия, φ
ln_phi	ln(ϕ)	0.48121	

Случайные числа

В некоторых разработках часто необходимо генерировать случайные числа. В научных вычислениях может потребоваться запустить большое количество симуляций

на основе случайных чисел. Такие числа должны эмулировать лотерею из случайных процессов с определенными характеристиками, например из пуассоновского или нормального распределения. Часто бывает нужно, чтобы эти симуляции повторялись, поэтому код, ответственный за генерацию случайности — *механизм случайных чисел*, — должен давать одинаковые выходные данные при одинаковых входных данных. Такие механизмы случайных чисел иногда называют механизмами *псевдослучайных* чисел.

В криптографии могут потребоваться случайные числа, чтобы защитить информацию. В таких условиях для кого-то должно быть практически невозможно получить подобный поток случайных чисел, поэтому случайное использование механизмов псевдослучайных чисел часто серьезно подрывает криптосистему, защищенную в противном случае.

По этим и другим причинам *никогда не стоит пытаться создать свой собственный генератор случайных чисел*. Создание правильного генератора случайных чисел на удивление сложно. Слишком легко ввести шаблоны в генератор случайных чисел, которые могут иметь неприятные и трудные для диагностики побочные эффекты в системах, которые используют ваши случайные числа в качестве входных данных.

ПРИМЕЧАНИЕ

Если вас интересует генерация случайных чисел, обратитесь к главе 2 книги «Stochastic Simulation» Брайана Рипли (Brian D. Riple) для научного применения и главе 2 книги «Serious Cryptography» Жан-Филиппа Аумассона (Jean-Philippe Aumasson) для криптографического применения.

Если вы ищете случайные числа, обратите внимание на библиотеки Random, доступные в stdlib в заголовке `<random>` или в Boost в заголовках `<boost / math / ...>`.

Механизм случайных чисел

Механизм случайных чисел генерируют случайные биты. В Boost и stdlib существует огромное множество примеров. Вот общее правило: если нужны повторяемые псевдослучайные числа, рассмотрите возможность использования механизма вихря Мерсенна `std::mtt19937_64`. Если нужны криптографически безопасные случайные числа, рассмотрите возможность использования `std::random_device`.

Вихрь Мерсенна обладает некоторыми желательными статистическими свойствами для моделирования. Его конструктору предоставляется целочисленное начальное значение, которое полностью определяет последовательность случайных чисел. Все механизмы случайных чисел являются функциональными объектами; чтобы получить случайное число, используйте вызов функции `operator()`. В листинге 12.29 показано, как создать механизм вихря Мерсенна с начальным числом 91586 и запустить полученный механизм трижды.

Листинг 12.29. mt19937_64 — механизм псевдослучайных чисел

```
#include <random>
TEST_CASE("mt19937_64 is pseudorandom") {
  std::mt19937_64 mt_engine{ 91586 }; ❶
  REQUIRE(mt_engine() == 8346843996631475880); ❷
  REQUIRE(mt_engine() == 2237671392849523263); ❸
  REQUIRE(mt_engine() == 7333164488732543658); ❹
}
```

Здесь создается механизм вихря Мерсенна `mt19937_64` с началом в 91586 ❶. Поскольку это псевдослучайный механизм, вы гарантированно получите одну и ту же последовательность случайных чисел ❷ ❸ ❹ каждый раз. Эта последовательность полностью определяется началом.

В листинге 12.30 показано, как создать `random_device` и вызвать его для получения криптографически безопасного случайного значения.

Листинг 12.30. random_device — это функциональный объект

```
TEST_CASE("std::random_device is invocable") {
  std::random_device rd_engine{}; ❶
  REQUIRE_NOTHROW(rd_engine()); ❷
}
```

`random_device` создается, используя конструктор по умолчанию ❶. Результирующий объект `rd_engine` ❷ можно вызвать, но стоит рассматривать объект как непрозрачный. В отличие от вихря Мерсенна в листинге 12.29, `random_device` непредсказуем по своей конструкции.

ПРИМЕЧАНИЕ

Поскольку компьютеры по своей конструкции являются детерминированными, `std::random_device` не может дать каких-либо серьезных гарантий в отношении криптографической безопасности.

Распределение случайных чисел

Распределение случайных чисел — это математическая функция, которая отображает число в плотности вероятности. Идея заключается в том, что при взятии выборки из случайных чисел, которые имеют конкретное распределение, и при построении графиков относительных частот значений выборок этот график будет выглядеть как распределение.

Распределения делятся на две большие категории: *дискретные* и *непрерывные*. Простая аналогия состоит в том, что дискретные распределения отображают интегральные значения, а непрерывные распределения отображают значения с плавающей точкой.

Большинство дистрибутивов принимают параметры настройки. Например, нормальное распределение — это непрерывное распределение, которое принимает два

параметра: среднее значение и дисперсию. Его плотность имеет знакомую форму колокола с центром вокруг среднего значения, как показано на рис. 12.1. Дискретное равномерное распределение является распределением случайных чисел, которое присваивает равные вероятности числам между некоторым минимумом и максимумом. Его плотность выглядит идеально плоской в диапазоне от минимального до максимального, как показано на рис. 12.2.

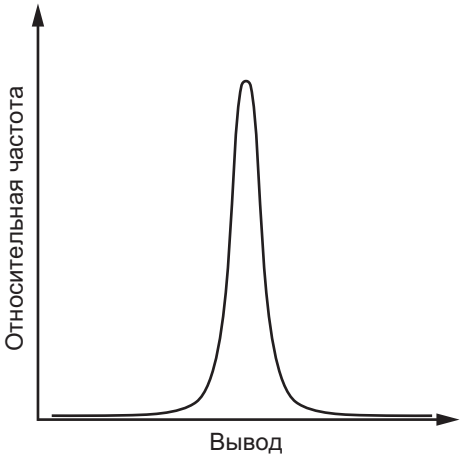


Рис. 12.1. Представление функции плотности вероятности нормального распределения

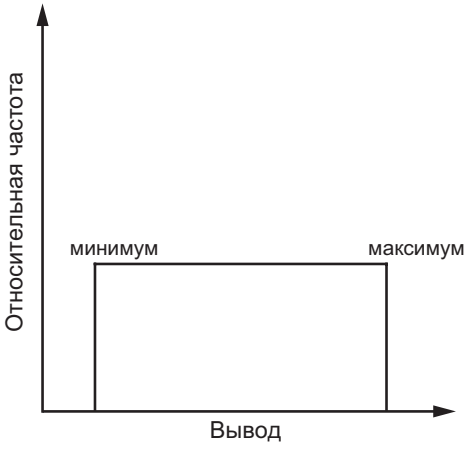


Рис. 12.2. Представление функции плотности вероятности равномерного распределения

Можно легко генерировать случайные числа из общих статистических распределений, таких как равномерное и нормальное, используя одну и ту же библио-

теку Random в stdlib. Каждое распределение принимает некоторые параметры в конструкторе, соответствующие параметрам базового распределения. Чтобы вывести случайную переменную из распределения, используется вызов функции `operator()` и передается экземпляр механизма случайных чисел, такой как вихрь Мерсенна.

`std::uniform_int_distribution` — это шаблон класса, доступный в заголовке `<random>`, который принимает один параметр шаблона, соответствующий типу, который нужно получить при выведении из распределения, например `int`.

Определяется минимум и максимум равномерного распределения, передавая их в качестве параметров конструктора. Каждое число в диапазоне имеет равную вероятность. Это, пожалуй, самое распространенное распределение, возникающее в общем контексте разработки программного обеспечения.

В листинге 12.31 показано, как взять миллион чисел из равномерного распределения с минимумом 1 и максимумом 10 и вычислить среднее значение выборки.

Листинг 12.31. Параметр uniform_int_distribution имитирует вывод из дискретного равномерного распределения

```
TEST_CASE("std::uniform_int_distribution produces uniform ints") {
  std::mt19937_64 mt_engine{ 102787 }; ❶
  std::uniform_int_distribution<int> int_d{ 0, 10 }; ❷
  const size_t n{ 1'000'000 }; ❸
  int sum{}; ❹
  for (size_t i{}; i < n; i++)
    sum += int_d(mt_engine); ❺
  const auto sample_mean = sum / double{ n }; ❻
  REQUIRE(sample_mean == Approx(5).epsilon(.1)); ❼
}
```

Создается вихрь Мерсенна с начальным значением 102787 ❶, а затем создается `uniform_int_distribution` с минимальным значением 0 и максимальным значением 10 ❷. Затем инициализируется переменная `n` для хранения числа итераций ❸ и переменная для хранения суммы всех равномерных случайных величин ❹. В цикле отрисовываются случайные величины из равномерного распределения с помощью `operator()`, передавая экземпляр вихря Мерсенна ❺.

Среднее дискретное равномерное распределение — это минимум плюс максимум, деленный на 2. Здесь `int_d` имеет среднее значение 5. Можно вычислить среднее значение выборки, разделив сумму на количество выборок `n` ❻. С высокой достоверностью можно утверждать, что этот `sample_mean` равен приблизительно 5 ❼.

Неполный список распределений случайных чисел

Таблица 12.14 содержит неполный список распределений случайных чисел в `<random>`, их параметры шаблона по умолчанию и параметры их конструктора.

Таблица 12.14. Распределение случайных чисел в <random>

Распределение	Примечания
uniform_int_distribution<int>{ **min, max** }	Дискретное равномерное распределение с минимальным **min** и максимальным **max** значениями
uniform_real_distribution<double>{ **min, max** }	Непрерывное равномерное распределение с минимальным min и максимальными max значениями
normal_distribution<double>{ **m, s** }	Нормальное распределение со средним m и стандартным отклонением s. Обычно используется для моделирования добавочного результата многих независимых случайных величин. Также называется гауссовым распределением
lognormal_distribution<double>{ **m, s** }	Логически нормальное распределение со средним m и стандартным отклонением s Обычно используется для моделирования мультипликативного результата многих независимых случайных величин. Также называется распределением Гальтона
chi_squared_distribution<double>{ **n** }	Распределение хи-квадрат со степенями свободы n. Обычно используется в выводной статистике
cauchy_distribution<double>{ **a, b** }	Распределение Коши с параметром местоположения a и параметром масштаба b. Используется в физике. Также называется распределением Лоренца
fisher_f_distribution<double>{ **m, n** }	F-распределение со степенями свободы m и n. Обычно используется в выводной статистике. Также называется распределением Снедекора
student_t_distribution<double>{ **n** }	T-распределение со степенями свободы n. Обычно используется в выводной статистике. Также называется T-распределением Стьюдента
bernoulli_distribution{ **p** }	Распределение Бернулли с вероятностью успеха p. Обычно используется для моделирования одного логического результата
binomial_distribution<int>{ **n, p** }	Биномиальное распределение с n испытаниями и вероятностью успеха p. Обычно используется для моделирования количества успехов при отборе проб с заменой в серии экспериментов Бернулли
geometric_distribution<int>{ **p** }	Геометрическое распределение с вероятностью успеха p. Обычно используется для моделирования количества отказов, произошедших до первого успеха в серии экспериментов Бернулли
poisson_distribution<int>{ **m** }	Распределение Пуассона со средним m Обычно используется для моделирования количества событий, происходящих за фиксированный интервал времени

Распределение	Примечания
`exponential_ distribution<double>{ l }`	Экспоненциальное распределение со средним `1/l`, где `l` известно как параметр лямбда. Обычно используется для моделирования промежутка времени между событиями в процессе Пуассона
`gamma_distribution<double>{ a, b }`	Гамма-распределение с параметром формы `a` и параметром масштаба `b`. Обобщение экспоненциального распределения и распределения хи-квадрат
`weibull_distribution<double>{ k, l }`	Распределение Вейбулла с параметром формы `k` и параметром масштаба `l` Обычно используется для моделирования времени до отказа
`extreme_value_ distribution<double>{ a, b }`	Распределение экстремальных значений с параметром местоположения `a` и параметром масштаба `b`. Обычно используется для моделирования максимумов независимых случайных величин. Также называется распределением Гамбеля `I` типа

ПРИМЕЧАНИЕ

Boost Math предлагает больше распределений случайных чисел в серии заголовков `<boost/math/...>`, например бета, гипергеометрическое, логистическое и обратное нормальное распределения.

Числовые ограничения

Stdlib предлагает шаблон класса `std::numeric_limits` в заголовке `<limit>`, чтобы предоставить информацию времени компиляции о различных свойствах для арифметических типов. Например, если нужно идентифицировать наименьшее конечное значение для данного типа `T`, можно использовать статическую функцию-член `std::numeric_limits<T>::min()` для его получения.

Листинг 12.32 показывает, как использовать `min` для облегчения потери значимости.

Листинг 12.32. Использование std::numeric_limits<T>::min() для облегчения потери значимости int

```
#include <limits>
TEST_CASE("std::numeric_limits::min provides the smallest finite value.") {
  auto my_cup = std::numeric_limits<int>::min(); ❶
  auto underfloweth = my_cup - 1; ❷
  REQUIRE(my_cup < underfloweth); ❸
}
```

Хотя во время вывода основные компиляторы создают код, который проходит тест, эта программа содержит неопределенное поведение.

Сначала переменная `my_cup` устанавливается равной наименьшему возможному значению `int`, используя `std::numeric_limits<int>::min()` ❶. Затем намеренно вызывается потеря значения, вычитая 1 из `my_cup` ❷. Поскольку `my_cup` — это минимальное значение, которое может принимать `int`, `my_cup` приводит к потере значимости. Это вызывает ненормальную ситуацию, когда `underfloweth` больше, чем `my_cup` ❸, даже если `underfloweth` инициализирована путем вычитания из `my_cup`.

ПРИМЕЧАНИЕ

Такие скрытые потери были причиной неисчислимого числа уязвимостей безопасности ПО. Не полагайтесь на это неопределенное поведение!

Многие статические функции-члены и постоянные-члены доступны в `std::numeric_limits`. В таблице 12.15 перечислены некоторые из наиболее распространенных.

Таблица 12.15. Некоторые общие константы-члены в `std::numeric_limits`

Операция	Примечания
`numeric_limits<T>::is_signed`	`true`, если `T` — знаковое
`numeric_limits<T>::is_integer`	`true`, если `T` — целое
`numeric_limits<T>::has_infinity`	Определяет, может ли `T` кодировать бесконечное значение. (Обычно все типы с плавающей точкой имеют бесконечное значение, а целочисленные типы — нет)
`numeric_limits<T>::digits10`	Определяет количество цифр, которые `T` может предоставить
`numeric_limits<T>::min()`	Возвращает наименьшее значение `T`
`numeric_limits<T>::max()`	Возвращает наибольшее значение `T`

ПРИМЕЧАНИЕ

Boost Integer предоставляет некоторые дополнительные возможности для интроспективного анализа целочисленных типов, например определяет самое высокое или наименьшее целое число или наименьшее целое число, по крайней мере с `N` битами.

Numeric Conversion в Boost

Boost предоставляет библиотеку Numeric Conversion (преобразования чисел), которая содержит набор инструментов для преобразования числовых объектов. Шаблон класса `boost::numeric::converter` в заголовке `<boost/numeric/translation/converter.hpp>` инкапсулирует код для выполнения конкретного преобразования чисел из одного типа в другой.

Нужно предоставить два параметра шаблона: целевой тип `T` и исходный тип `S`. Можно указать числовой преобразователь, который принимает `double` и преобразует его в тип `int` с простым псевдонимом типа `double_to_int`:

```
#include <boost/numeric/conversion/converter.hpp>
using double_to_int = boost::numeric::converter<int❶, double❷>;
```

Для преобразования с новым псевдонимом типа `double_to_int` есть несколько вариантов. Во-первых, можно использовать его статический метод `convert`, который принимает `double` ❷ и возвращает `int` ❶, как показано в листинге 12.33.

Листинг 12.33. boost::numeric::converter предлагает статический метод convert

```
TEST_CASE("boost::numeric::converter offers the static method convert") {
  REQUIRE(double_to_int::convert(3.14159) == 3);
}
```

Здесь просто вызывается метод `convert` со значением 3.14159, которое `boost::convert` преобразует в 3.

Поскольку `boost::convert` предоставляет функцию вызова `operator()`, можно создать объект функции `double_to_int` и использовать его для преобразования, как показано в листинге 12.34.

Листинг 12.34. boost::numeric::converter реализует operator()

```
TEST_CASE("boost::numeric::converter implements operator()") {
  double_to_int dti; ❶
  REQUIRE(dti(3.14159) == 3); ❷
  REQUIRE(double_to_int{}(3.14159) == 3); ❸
}
```

Создается объект функции `double_to_int` с именем `dti` ❶, который вызывается с тем же аргументом 3.14159 ❷, что и в листинге 12.33. Результат тот же. Также есть возможность создать временный объект функции и напрямую использовать `operator()`, что дает идентичные результаты ❸.

Основным преимуществом использования `boost::numeric::converter` вместо таких альтернатив, как `static_cast`, является проверка границ во время выполнения. Если преобразование вызовет переполнение, `boost::numeric::converter` выбросит `boost::numeric::positive_overflow` или `boost::numeric::negative_overflow`. Листинг 12.35 показывает это поведение при попытке преобразовать очень большой `double` в `int`.

Листинг 12.35. boost::numeric::converter проверяет переполнение

```
#include <limits>
TEST_CASE("boost::numeric::converter checks for overflow") {
  auto yuge = std::numeric_limits<double>::max(); ❶
  double_to_int dti; ❷
  REQUIRE_THROWS_AS(dti(yuge)❸, boost::numeric::positive_overflow❹);
}
```

numeric_limits используется для получения значения yuge ❶. Создается double_to_int конвертер ❷, который используется для попытки преобразования yuge в int ❸. Это вызывает исключение positive_overflow, потому что значение слишком велико для хранения ❹.

Можно настроить поведение преобразования boost::numeric::converter с помощью параметров шаблона. Например, можно настроить обработку переполнения, чтобы вызвать пользовательское исключение или выполнить какую-либо другую операцию. Также можно настроить поведение округления таким образом, чтобы вместо усечения десятичного числа из значения с плавающей точкой выполнялось пользовательское округление. Подробности смотрите в документации по ускоренному преобразованию чисел.

Если вас устраивает стандартное поведение boost::numeric::converter, можно использовать шаблон функции boost::numeric_cast в качестве сокращения. Этот шаблон функции принимает один параметр шаблона, соответствующий целевому типу преобразования, и один аргумент, соответствующий номеру источника. В листинге 12.36 приведено обновление листинга 12.35, в котором вместо этого используется boost::numeric _cast.

Листинг 12.36. Шаблон функции boost::numeric_cast также выполняет проверку границ во время выполнения

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

TEST_CASE("boost::boost::numeric_cast checks overflow") {
  auto yuge = std::numeric_limits<double>::max(); ❶
  REQUIRE_THROWS_AS(boost::numeric_cast<int>(yuge), ❷
                    boost::numeric::positive_overflow ❸);
}
```

Как и прежде, numeric_limits используется для получения значения yuge ❶. При попытке преобразовать numeric_cast yuge в int ❷ вы получаете исключение positive_overflow, потому что значение слишком велико для хранения ❸.

ПРИМЕЧАНИЕ

Шаблон функции boost::numeric_cast является подходящей заменой для narrow_cast, который мы собирали вручную в листинге 6.6.

Рациональная арифметика во время компиляции

std::ratioв stdlib в заголовке <ratio> — это шаблон класса, который позволяет вычислять рациональную арифметику во время компиляции. Для std::ratio предоставляется два параметра шаблона: числитель и знаменатель. Это определяет новый тип, который можно использовать для вычисления рациональных выражений.

Вычисления выполняются во время компиляции с помощью `std::ratio` и методов метапрограммирования шаблонов. Например, чтобы умножить два типа `ratio`, можно использовать тип `std::ratio_multiply`, который принимает два типа `ratio` в качестве параметров шаблона. Можно извлечь числитель и знаменатель результата, используя статические переменные-члены для результирующего типа.

В листинге 12.37 показано, как умножить 10 на 2/3 во время компиляции.

Листинг 12.37. Рациональная арифметика времени компиляции с использованием std::ratio

```
#include <ratio>

TEST_CASE("std::ratio") {
  using ten = std::ratio<10, 1>; ❶
  using two_thirds = std::ratio<2, 3>; ❷
  using result = std::ratio_multiply<ten, two_thirds>; ❸
  REQUIRE(result::num == 20); ❹
  REQUIRE(result::den == 3); ❺
}
```

Типы `std::ratio ten` ❶ и `two_thirds` ❷ объявляются как псевдонимы типов. Чтобы вычислить произведение `ten` и `two_thirds`, снова объявляется другой тип, `result`, с использованием шаблона `std::ratio_multiply` ❸. Используя статические члены `num` и `den`, можно извлечь результат, 20/3 ❹ ❺.

Конечно, всегда лучше выполнять вычисления во время компиляции, а не во время выполнения, когда это возможно. Программы будут более эффективными, потому что при запуске им потребуется меньше вычислений.

Неполный список распределений случайных чисел

Таблица 12.16 содержит неполный список операций, предоставляемых библиотекой `<ratio>`в stdlib.

Таблица 12.16. Неполный список операций, доступных в `<ratio>`

Операция	Примечания
`ratio_add<r1, r2>`	Складывает `r1` и `r2`
`ratio_subtract<r1, r2>`	Вычитает `r1` из `r2`
`ratio_multiply<r1, r2>`	Умножает `r1` на `r2`
`ratio_divide<r1, r2>`	Делит `r1` на `r2`
`ratio_equal<r1, r2>`	Проверяет равенство `r1` и `r2`
`ratio_not_equal<r1, r2>`	Проверяет неравенство `r1` и `r2`
`ratio_less<r1, r2>`	Проверяет, меньше ли `r1`, чем `r2`
`ratio_greater<r1, r2>`	Проверяет, больше ли `r1`, чем `r2`

Продолжение ⇩

Таблица 12.16 *(продолжение)*

Операция	Примечания
`ratio_less_equal<r1, r2>`	Проверяет, что `r1` меньше или равен `r2`
`ratio_greater_equal<r1, r2>`	Проверяет, что `r1` больше или равен `r2`
`Micro`	Литерал: `ratio<1, 1000000>`
`Milli`	Литерал: `ratio<1, 1000>`
`Centi`	Литерал: `ratio<1, 100>`
`Deci`	Литерал: `ratio<1, 10>`
`Deca`	Литерал: `ratio<10, 1>`
`Hecto`	Литерал: `ratio<100, 1>`
`Kilo`	Литерал: `ratio<1000, 1>`
`Mega`	Литерал: `ratio<1000000, 1>`
`Giga`	Литерал: `ratio<1000000000, 1>`

Итоги

В этой главе вы рассмотрели ряд небольших и простых специализированных утилит, которые обслуживают общие потребности программирования. Структуры данных, такие как `tribool`, `optional`, `pair`, `tuple`, `any` и `variant`, обрабатывают многие обычные сценарии, в которых нужно хранить объекты в общей структуре. В следующих главах некоторые из этих структур данных будут повторно встречаться по всему stdlib. Вы также узнали о дате/времени и числовых/математических средствах. Эти библиотеки реализуют очень специфическую функциональность, но при наличии таких требований эти библиотеки неоценимы.

Упражнения

12.1. Переопределите `narrow_cast` в листинге 6.6, чтобы возвратить `std::optional`. Если приведение приведет к сужающемуся преобразованию, верните пустой `optional` параметр, а не выбрасывайте исключение. Напишите юнит-тест, который гарантирует, что решение работает.

12.2. Реализуйте программу, которая генерирует случайные буквенно-цифровые пароли и записывает их в консоль. Можно сохранить алфавит возможных символов в `char[]` и использовать дискретное равномерное распределение с минимумом 0 и максимумом последнего индекса алфавитного массива. Используйте криптографически безопасный механизм случайных чисел.

Что еще почитать?

- Международный стандарт ИСО/МЭК (2017) — Язык программирования C++ (Международная организация по стандартизации; Женева, Швейцария; isocpp.org/std/the-standard/)
- «The Boost C++ Libraries», 2nd Edition, Boris Schäling (XML Press, 2014)
- «Стандартная библиотека C++. Справочное руководство», 2-е издание, Николай М. Джосаттис (Вильямс, 2017)

13 Контейнеры

Исправление ошибок в `std::vector` — это в равных частях восторг (это самая лучшая структура данных) и ужас (если я все испорчу, мир рухнет).

Стефан Т. Лававей (главный разработчик библиотеки Visual C++), твит от 22 августа 2016 года

Стандартная библиотека шаблонов (STL) — это часть stdlib, которая предоставляет контейнеры и алгоритмы для управления ими, а итераторы служат интерфейсом между ними. В следующих трех главах вы узнаете больше о каждом из этих компонентов.

Контейнер — это специальная структура данных, которая хранит объекты организованным образом, следуя определенным правилам доступа. Существует три вида контейнеров:

- контейнеры последовательности хранят элементы последовательно, как в массиве;
- ассоциативные контейнеры хранят отсортированные элементы;
- неупорядоченные ассоциативные контейнеры хранят хешированные объекты.

Ассоциативные и неупорядоченные ассоциативные контейнеры обеспечивают быстрый поиск отдельных элементов. Все контейнеры являются RAII-обертками вокруг содержащихся в них объектов, поэтому они управляют длительностью хранения и временем жизни принадлежащих им элементов. Кроме того, каждый контейнер предоставляет некоторый набор функций-членов, которые выполняют различные операции над коллекцией объектов.

Современные программы на C++ постоянно используют контейнеры. Какой контейнер вы выберете для конкретного приложения, зависит от требуемых операций, характеристик содержащихся объектов и эффективности при определенных шаблонах доступа. В этой главе рассматривается огромный контейнерный ландшафт между STL и Boost. Поскольку в этих библиотеках много контейнеров, мы изучим только самые популярные из них.

Контейнеры последовательностей

Контейнеры последовательностей — это контейнеры STL, которые позволяют использовать последовательный доступ к элементам.

То есть можно начать с одного конца контейнера и перейти к другому. Но это единственная их схожесть, контейнеры последовательности — разнообразная и пестрая команда. Некоторые контейнеры имеют фиксированную длину; другие могут уменьшаться и расти в зависимости от потребностей программы. Некоторые позволяют индексировать непосредственно в контейнер, тогда как в других можно только последовательно получать доступ к элементам. Кроме того, каждый контейнер последовательности имеет уникальные рабочие характеристики, которые делают его желательным в одних ситуациях и нежелательным в других.

Работа с контейнерами последовательностей должна быть интуитивно понятной, поскольку вы знакомы с примитивами со времен «Массивов», где вы видели встроенный массив или массив в стиле C — `T[]`. Вы начнете обзор контейнеров последовательностей с встроенного более сложного, более крутого младшего брата массива — `std::array`.

Массивы

STL предоставляет `std::array` в заголовке `<array>`. `array` — это последовательный контейнер, содержащий непрерывную последовательность элементов фиксированного размера. Он сочетает в себе высокую производительность и эффективность встроенных массивов с современными удобствами поддержки построения/назначения копирования/перемещения, знания своего собственного размера, обеспечения доступа к элементу с проверкой границ и других расширенных функций.

Следует использовать `array` вместо встроенных массивов практически во всех ситуациях. Он поддерживает почти все те же шаблоны использования, что и `operator[]` для доступа к элементам, поэтому не так много ситуаций, в которых вместо этого понадобится встроенный массив.

ПРИМЕЧАНИЕ

Boost также предлагает `boost::array` в Boost Array в заголовке `<boost/array.hpp>`. Не нужно использовать версию Boost, если у вас нет очень старой цепочки инструментов C++.

Создание

Шаблон класса `<T, S>` принимает два параметра шаблона:

- содержащийся тип `T`;
- фиксированный размер массива `S`.

Можно создать `array` и встроенные массивы, используя те же правила. Чтобы суммировать эти правила из «Массивов» на с. 100, предпочтительным методом является использование фигурной инициализации для создания массива. Инициализация в фигурных скобках заполняет массив значениями, содержащимися в фигурных скобках, и заполняет оставшиеся элементы нулями. Если пропустить инициализационные скобки, массив будет содержать неинициализированные значения в зависимости от длительности его хранения. Листинг 13.1 показывает инициализацию в скобках с несколькими объявлениями массива.

Листинг 13.1. Инициализация std::array. Вы можете получить предупреждения компилятора из REQUIRE(local_array[0]!= 0); ❹, так как local_array имеет неинициализированные элементы

```
#include <array>

std::array<int, 10> static_array{}; ❶

TEST_CASE("std::array") {
  REQUIRE(static_array[0] == 0); ❷

  SECTION("uninitialized without braced initializers") {
    std::array<int, 10> local_array; ❸
    REQUIRE(local_array[0] != 0); ❹
  }

  SECTION("initialized with braced initializers") {
    std::array<int, 10> local_array{ 1, 1, 2, 3 };❺
    REQUIRE(local_array[0] == 1);
    REQUIRE(local_array[1] == 1);
    REQUIRE(local_array[2] == 2);
    REQUIRE(local_array[3] == 3);
    REQUIRE(local_array[4] == 0); ❻
  }
}
```

Объявляется массив из 10 `int`-объектов, который называется `static_array`, со статической длительностью хранения ❶. Вы не использовали фигурную инициализацию, но ее элементы все равно инициализируются нулями ❷, благодаря правилам инициализации, описанным в «Массивах» на с. 100.

Затем вы пытаетесь объявить другой массив из 10 `int`-объектов, на этот раз с автоматической длительностью хранения ❸. Поскольку фигурная инициализация не использовалась, `local_array` содержит неинициализированные элементы (которые с крайне низкой вероятностью равны нулю ❹).

Наконец, используется фигурная инициализация, чтобы объявить другой массив и заполнить первые четыре элемента ❺. Все остальные элементы заполняются нулями ❻.

Доступ к элементам

Три основных метода, с помощью которых можно получить доступ к произвольным элементам массива:

- `operator[]`;
- `at`;
- `get`.

`operator[]` и `at` принимают единственный аргумент `size_t`, соответствующий индексу нужного элемента. Разница между ними заключается в проверке границ: если аргумент `index` выходит за пределы, `at` вызовет исключение `std::out_of_range`, тогда как `operator[]` вызовет неопределенное поведение. Шаблон функции `get` принимает параметр шаблона той же спецификации. Поскольку это шаблон, индекс должен быть известен во время компиляции.

ПРИМЕЧАНИЕ

Вспомните из «Типа `size_t`» на с. 98, что объект `size_t` гарантирует, что его максимальное значение является достаточным для представления максимального размера в байтах всех объектов. По этой причине `operator[]` и `at` принимают `size_t`, а не `int`, что не дает такой гарантии.

Основным преимуществом использования `get` является то, что вы получаете проверку границ во время компиляции, как показано в листинге 13.2.

Листинг 13.2. Доступ к элементам массива. Раскомментирование `// fib[4] = 5;` ❹ вызовет неопределенное поведение, тогда как раскомментирование `// std::get(fib);` ❿ приведет к ошибке компиляции

```
TEST_CASE("std::array access") {
   std::array<int, 4> fib{ 1, 1, 0, 3}; ❶

  SECTION("operator[] can get and set elements") {
    fib[2] = 2; ❷
    REQUIRE(fib[2] == 2); ❸
    // fib[4] = 5; ❹
  }

  SECTION("at() can get and set elements") {
    fib.at(2) = 2; ❺
    REQUIRE(fib.at(2) == 2); ❻
    REQUIRE_THROWS_AS(fib.at(4), std::out_of_range); ❼
  }
  SECTION("get can get and set elements") {
    std::get<2>(fib) = 2; ❽
    REQUIRE(std::get<2>(fib) == 2); ❾
    // std::get<4>(fib); ❿
  }
}
```

Объявляется массив длины 4 с именем `fib` ❶. Используя `operator[]` ❷, можно устанавливать элементы и извлекать их ❸. Запрещенная запись, которую вы закомментировали, может привести к неопределенному поведению; нет проверки границ при использовании `operator[]`❹.

Можно использовать `at` для одной и той же операции чтения ❺ и записи ❻, и можно безопасно выполнять операцию за пределами допустимой границы благодаря проверке границ ❼.

Наконец, можно использовать `std::get` для задания ❽ и получения ❾ элементов. Элемент `get` также выполняет проверку границ, поэтому код`// std::ge <4>(fib);` ❿ не удастся скомпилировать, если он не закомментирован.

Также заданы методы `front` и `back`, которые возвращают ссылки на первый и последний элементы массива. Вы получите неопределенное поведение, если вызовете один из этих методов в массиве, который имеет нулевую длину, как показано в листинге 13.3.

Листинг 13.3. Использование вспомогательных методов front и back в std::array

```
TEST_CASE("std::array has convenience methods") {
  std::array<int, 4> fib{ 0, 1, 2, 0 };

  SECTION("front") {
    fib.front() = 1; ❶
    REQUIRE(fib.front() == 1); ❷
    REQUIRE(fib.front() == fib[0]); ❸
  }

  SECTION("back") {
    fib.back() = 3; ❹
    REQUIRE(fib.back() == 3); ❺
    REQUIRE(fib.back() == fib[3]); ❻
  }
}
```

Можно использовать методы `front` и `back` для установки ❶ ❹ и получения ❷ ❺ первого и последнего элементов массива. Конечно, `fib[0]` идентичен `fib.front()` ❸, а `fib[3]` идентичен `fib.back()`❻. Методы `front()`и `back()` являются просто вспомогательными. Кроме того, при написании общего кода некоторые контейнеры будут предлагать `front` и `back`, но не `operator[]`, поэтому лучше использовать методы `front` и `back`.

Модель хранения

`array` не делает выделения памяти; скорее как встроенный массив он содержит все свои элементы. Это означает, что копирование, как правило, будет затратным, потому что каждый составляющий элемент должен быть скопирован. Перемещения могут быть затратными, в зависимости от того, имеет ли базовый тип `array` также конструктор переноса и присваивание переноса, которые относительно незатратны.

Каждый `array` — это просто встроенный массив под капотом. Фактически можно извлечь указатель на первый элемент `array`, используя четыре различных метода.

- Один из лучших методов — метод `data`. Как было сказано, он возвращает указатель на первый элемент.
- Другие три метода включают использование оператора вычисления адреса & на первом элементе, который можно получить с помощью `operator[]`, `at` и `front`.

Стоит использовать `data`. Если `array` пуст, подходы на основе адреса будут возвращать неопределенное поведение.

В листинге 13.4 показано, как получить указатель, используя эти четыре метода.

Листинг 13.4. Получение указателя на первый элемент std::array

```
TEST_CASE("We can obtain a pointer to the first element using") {
  std::array<char, 9> color{ 'o', 'c', 't', 'a', 'r', 'i', 'n', 'e' };
  const auto* color_ptr = color.data(); ❶

  SECTION("data") {
    REQUIRE(*color_ptr == 'o'); ❷
  }
  SECTION("address-of front") {
    REQUIRE(&color.front() == color_ptr); ❸
  }
  SECTION("address-of at(0)") {
    REQUIRE(&color.at(0) == color_ptr); ❹
  }
  SECTION("address-of [0]") {
    REQUIRE(&color[0] == color_ptr); ❺
  }
}
```

После инициализации `array color` вы получаете указатель на первый элемент, букву `o`, используя метод `data` ❶. При разыменовании результирующего `color_ptr` вы получаете букву `o`, как и ожидалось ❷. Этот указатель идентичен указателю, полученному из подходов вычисления-адреса-плюс-`front` ❸, `-at` ❹ и `-operator[]` ❺.

Чтобы закончить обсуждение массивов, можно запросить размер `array`, используя методы `size` или `max_size`. (Они идентичны для `array`.) Поскольку `array` имеет фиксированный размер, значения этого метода являются статическими и известны во время компиляции.

Краткий курс по итераторам

Интерфейс между контейнерами и алгоритмами — это итератор. Итератор — это тип, который знает внутреннюю структуру контейнера и предоставляет простые, похожие на указатель операции с элементами контейнера. Глава 14 полностью посвящена итераторам, но необходимо знать основы, чтобы вы могли изучить, как

использовать итераторы для управления контейнерами и как контейнеры предоставляют итераторы пользователям.

Итераторы бывают разных типов, но все они поддерживают по крайней мере следующие операции:

- получить текущий элемент (`operator*`);
- перейти к следующему элементу (`operator++`);
- назначить итератор равным другому итератору (`operator=`).

Можно извлечь итераторы из всех контейнеров STL (включая массив), используя методы `begin` и `end`. Метод `begin` возвращает итератор, указывающий на первый элемент, а метод `end` возвращает указатель на элемент после последнего элемента. На рис. 13.1 показано, как итераторы `begin` и `end` указывают на массив из трех элементов.

Рис. 13.1. Полуоткрытый диапазон в массиве из трех элементов

Расположение на рис. 13.1, где `end()` указывает на место после последнего элемента, называется *полуоткрытым диапазоном*. Поначалу это может показаться нелогичным — почему бы не иметь закрытый диапазон, в котором `end()` указывает на последний элемент, — но у полуоткрытого диапазона есть некоторые преимущества. Например, если контейнер пуст, метод `begin()` вернет то же значение, что и `end()`. Это позволяет знать, что независимо от того, является ли контейнер пустым, если итератор равен `end()`, контейнер пройден.

В листинге 13.5 показано, что происходит с итераторами с полуоткрытым диапазоном и пустыми контейнерами.

Листинг 13.5. При пустом массиве итератор begin равен итератору end

```
TEST_CASE("std::array begin/end form a half-open range") {
  std::array<int, 0> e{}; ❶
  REQUIRE(e.begin()❷ == e.end()❸);
}
```

Здесь создается пустой массив `e` ❶ и итераторы `begin` ❷ и `end` ❸ равны друг другу.

В листинге 13.6 показано, как использовать итераторы для выполнения операций с указателями над непустым массивом.

`array easy_as` содержит элементы 1, 2 и 3 ❶. `begin` вызывается для `easy_as`, чтобы получить итератор `iter`, указывающий на первый элемент ❷. Оператор разыменования возвращает первый элемент 1, потому что это первый элемент в `array` ❸. Затем

увеличивается `iter`, чтобы он указывал на следующий элемент ❹. Вы продолжаете в том же духе, пока не достигнете последнего элемента ❺. Увеличение указателя в последний раз приводит на место после последнего элемента ❻, поэтому он равен `easy_as.end()`, указывая, что массив пройден ❼.

Листинг 13.6. Основные операции с итератором массива

```
TEST_CASE("std::array iterators are pointer-like") {
  std::array<int, 3> easy_as{ 1, 2, 3 }; ❶
  auto iter = easy_as.begin(); ❷
  REQUIRE(*iter == 1); ❸
  ++iter; ❹
  REQUIRE(*iter == 2);
  ++iter;
  REQUIRE(*iter == 3); ❺
  ++iter; ❻
  REQUIRE(iter == easy_as.end()); ❼
}
```

Вспомните из «Выражений диапазона» на с. 303, что можно создавать свои собственные типы для использования в выражениях диапазона, предоставляя методы `begin` и `end`, как это реализовано в `FibonacciIterator` в листинге 8.29. Контейнеры уже делают всю эту работу за вас, то есть можно использовать любой контейнер STL как выражение диапазона. Листинг 13.7 показывает итерацию по `array`.

Листинг 13.7. Основанные на диапазоне циклы for и array

```
TEST_CASE("std::array can be used as a range expression") {
  std::array<int, 5> fib{ 1, 1, 2, 3, 5 };❶
  int sum{}; ❷
  for (const auto element : fib) ❸
    sum += element; ❹
  REQUIRE(sum == 12);
}
```

В коде инициализируются `array` ❶ и переменная `sum` ❷. Поскольку `array` является допустимым диапазоном, его можно использовать в цикле `for` ❸ на основе диапазона. Это позволяет накопить `sum` каждого `element` ❹.

Неполный список поддерживаемых операций

В таблице 13.1 приведен неполный список операций с `array`. В этой таблице `a`, `a1` и `a2` имеют тип `std::array<T, S>`, `t` имеет тип `T`, `S` является фиксированной длиной массива, а `i` имеет тип `size_t`.

ПРИМЕЧАНИЕ

Операции в таблице 13.1 функционируют как быстрые, достаточно полные ссылки. Для получения подробной информации обратитесь к свободно доступным онлайн-ссылкам cppreference.com и cplusplus.com, а также к главе 31 четвертого издания «Языка программирования C++» Бьёрна Страуструпа и главам 7, 8 и 12 2-го издания «Стандартной библиотеки C++. Справочного руководства» Николая М. Джосаттиса (Nicolai M. Josuttis).

Таблица 13.1. Неполный список операций `std::array`

Операция	Примечания
`array<T, S>{... }`	Выполняет фигурную инициализацию только что созданного массива
`~array`	Уничтожает все элементы, содержащиеся в массиве
`a1 = a2`	Выполняет присваивание копии всех членов `a1` с членами `a2`
`a.at(i)`	Возвращает ссылку на элемент `i` из `a`. Выдает `std::out_of_range`, если выходит за пределы `array`
`a[i]`	Возвращает ссылку на элемент `i` из `a`. Выдает неопределенное поведение, если выходит за пределы `array`
`get<i>(a)`	Возвращает ссылку на элемент `i` из `a`. Не компилируется, если выходит за пределы `array`
`a.front()`	Возвращает ссылку на первый элемент
`a.back()`	Возвращает ссылку на последний элемент
`a.data()`	Возвращает обычный указатель на первый элемент, если массив не пустой. Для пустых массивов возвращает действительный, но не разыменовываемый указатель
`a.empty()`	Возвращает `true`, если размер массива равен нулю; в противном случае — `false`
`a.size()`	Возвращает размер массива
`a.max_size()`	Идентичен `a.size()`
`a.fill(t)`	Выполняет присваивание копии `t` для каждого элемента
`a1.swap(a2)` `swap(a1, a2)`	Заменяет каждый элемент `a1` на элемент `a2`
`a.begin()`	Возвращает итератор, указывающий на первый элемент
`a.cbegin()`	Возвращает итератор `const`, указывающий на первый элемент
`a.end()`	Возвращает итератор, указывающий на место после последнего элемента
`a.cend()`	Возвращает итератор `const`, указывающий на место после последнего элемента
`a1 == a2`	Равенство выполняется, если все элементы равны

`a1 != a2`
`a1 > a2`
`a1 >= a2`
`a1 < a2`
`a1 <= a2`
	Сравнение больше/меньше, выполняется от первого элемента до последнего

Векторы

`std::vector` (вектор), доступный в заголовке STL `<vector>`, является последовательным контейнером, который содержит непрерывный ряд элементов динамического размера. `vector` управляет своим хранилищем динамически, не требуя внешней помощи от программиста.

`vector` является рабочей лошадкой стабильной структуры последовательных данных. При очень скромных издержках вы получаете значительную гибкость по сравнению с `array`. Кроме того, `vector` поддерживает почти все те же операции, что и массив, и добавляет множество других. Если у вас под рукой фиксированное количество элементов, следует строго рассмотреть `array`, потому что вы получите небольшое сокращение издержек по сравнению с вектором. Во всех других ситуациях самым лучшим последовательным контейнером является `vector`.

ПРИМЕЧАНИЕ

Библиотека Boost Container также содержит `boost::container::vector` в заголовке `<boost/container/vector.hpp>`.

Создание

Шаблон класса `std::vector<T, Allocator>` принимает два параметра шаблона. Первый — это содержащийся тип `T`, а второй — это тип распределителя `Allocator`, который является необязательным и по умолчанию имеет значение `std::allocator<T>`.

В создании векторов гораздо больше гибкости, чем у массивов. `vector` поддерживает пользовательские распределители, поскольку векторам необходимо выделять динамическую память. Можно по умолчанию создать `vector`, не содержащий элементов. Возможно, вы захотите создать пустой `vector`, чтобы заполнить его переменным числом элементов в зависимости от того, что происходит во время выполнения. Листинг 13.8 показывает создание `vector` по умолчанию и проверку, что он не содержит элементов.

Листинг 13.8. Вектор поддерживает создание по умолчанию

```
#include <vector>
TEST_CASE("std::vector supports default construction") {
  std::vector<const char*❶> vec; ❷
  REQUIRE(vec.empty()); ❸
}
```

В примере объявляется `vector`, содержащий элементы типа `const char*` ❶ с именем `vec`. Поскольку он был создан по умолчанию ❷, `vector` не содержит элементов, а метод `empty` возвращает `true` ❸.

Можно использовать фигурную инициализацию при создании `vector`. Подобно инициализации массива, она заполняет вектор указанными элементами, как показано в листинге 13.9.

Листинг 13.9. Вектор поддерживает фигурные инициализаторы

```
TEST_CASE("std::vector supports braced initialization ") {
    std::vector<int> fib{ 1, 1, 2, 3, 5 }; ❶
    REQUIRE(fib[4] == 5); ❷
}
```

Здесь создается `vector` с именем `fib` и используются фигурные инициализаторы ❶. После инициализации `vector` содержит пять элементов: 1, 1, 2, 3 и 5 ❷.

Если нужно заполнить `vector` множеством одинаковых значений, можно использовать один из *конструкторов заполнения*. Чтобы заполнить `vector` при создании, сначала передается `size_t`, соответствующий количеству элементов, которые нужно заполнить. При желании можно передать ссылку `const` на объект для копирования. Иногда нужно инициализировать все элементы одним и тем же значением, например, чтобы отслеживать количество связей с определенными индексами. Также может существовать `vector` пользовательского типа, который отслеживает состояние программы, и может потребоваться отслеживать такое состояние по индексу.

К сожалению, общее правило использования фигурной инициализации для создания объектов в этом случае нарушается. При использовании `vector` нужно использовать скобки для вызова этих конструкторов. Для компилятора `std::vector<int>{99, 100}` задается список инициализации с элементами 99 и 100, который создаст вектор с двумя элементами, 99 и 100. Что, если нужно заполнить вектор 99 копиями числа 100?

В общем, компилятор будет очень стараться рассматривать список инициализаторов как элементов для заполнения вектора. Можно попытаться запомнить правила (см. правило 7 «Эффективного использования C++» Скотта Мейерса) или просто взять на себя обязательство использовать скобки для конструкторов контейнеров в stdlib.

В листинге 13.10 показано общее правило списка инициализатора/фигурной инициализации для контейнеров STL.

Листинг 13.10. Вектор поддерживает фигурные инициализаторы и конструкторы заполнения

```
TEST_CASE("std::vector supports") {
  SECTION("braced initialization") {
    std::vector<int> five_nine{ 5, 9 }; ❶
    REQUIRE(five_nine[0] == 5); ❷
    REQUIRE(five_nine[1] == 9); ❸
  }
  SECTION("fill constructor") {
    std::vector<int> five_nines(5, 9); ❹
    REQUIRE(five_nines[0] == 9); ❺
    REQUIRE(five_nines[4] == 9); ❻
  }
}
```

В первом примере используется фигурная инициализация для построения вектора с двумя элементами ❶: 5 с индексом 0 ❷ и 9 с индексом 1 ❸. Во втором примере используются скобки для вызова конструктора заполнения ❹, который заполняет вектор пятью копиями числа 9, поэтому первый элемент ❺ и последний элемент ❻ равны 9.

ПРИМЕЧАНИЕ

Этот конфликт обозначений вызывает сожаление и не является результатом какого-то хорошо продуманного компромисса. Причины являются чисто историческими и связаны с обратной совместимостью.

Также можно создавать `vector` из полуоткрытого диапазона, передавая итераторы `begin` и `end` диапазона, который нужно скопировать. В различных контекстах программирования можно разделить подмножество некоторого диапазона и скопировать его в `vector` для дальнейшей обработки. Например, можно создать вектор, который копирует все элементы, содержащиеся в массиве, как показано в листинге 13.11.

Листинг 13.11. Создание вектора из диапазона

```
TEST_CASE("std::vector supports construction from iterators") {
  std::array<int, 5> fib_arr{ 1, 1, 2, 3, 5 }; ❶
  std::vector<int> fib_vec(fib_arr.begin(), fib_arr.end()); ❷
  REQUIRE(fib_vec[4] == 5); ❸
  REQUIRE(fib_vec.size() == fib_arr.size()); ❹
}
```

Создается массив `fib_arr` с пятью элементами ❶. Чтобы построить вектор `fib_vec` с элементами, содержащимися в `fib_arr`, вызываются методы `begin` и `end` для `fib_arr` ❷. Полученный `vector` имеет копии элементов массива ❸ и имеет одинаковый `size` ❹.

На высоком уровне можно представить действие этого конструктора как то, что он берет указатели на начало и конец некоторой целевой последовательности. Затем он копирует эту целевую последовательность.

Семантика копирования и переноса

С помощью `vector` вы получаете полную поддержку копирования/переноса/присваивания. Любая операция копирования в `vector` потенциально очень затратна, потому что это поэлементное или глубокое копирование. С другой стороны, операции переноса, как правило, выполняются очень быстро, поскольку содержащиеся в них элементы находятся в динамической памяти, а `vector`, из которого осуществляется перенос, может просто передать владение целевому вектору; нет необходимости перемещать содержащиеся элементы.

Доступ к элементам

Вектор поддерживает большинство тех же операций доступа к элементу, что и массив: `at`, `operator[]`, `front`, `back` и `data`.

Как и в случае с `array`, можно запросить количество элементов в `vector`, используя метод `size`. Возвращаемое значение этого метода может изменяться во время выполнения. Также можете определить, содержит ли `vector` какие-либо элементы, с помощью метода `empty`, который возвращает `true`, если `vector` не содержит элементов; в противном случае возвращается `false`.

Добавление элементов

Можно использовать различные методы для вставки элементов в `vector`. Если нужно заменить все элементы в `vector`, можно использовать метод `assign`, который берет список инициализации и заменяет все существующие элементы. При необходимости размер `vector` будет изменен, чтобы вместить больший список элементов, как показано в листинге 13.12.

Листинг 13.12. Метод assign в vector

```
TEST_CASE("std::vector assign replaces existing elements") {
  std::vector<int> message{ 13, 80, 110, 114, 102, 110, 101 }; ❶
  REQUIRE(message.size() == 7); ❷
  message.assign({ 67, 97, 101, 115, 97, 114 }); ❸
  REQUIRE(message[5] == 114); ❹
  REQUIRE(message.size() == 6); ❺
}
```

Здесь создается `vector` ❶ с семью элементами ❷. При назначении нового, меньшего списка инициализаторов ❸ все элементы заменяются ❹, а `size` вектора обновляется в соответствии с новым содержимым ❺.

Если нужно добавить один новый элемент в вектор, можно использовать метод `insert`, который ожидает два аргумента: итератор и элемент для вставки. Он вставит копию данного элемента непосредственно перед существующим элементом, на который указывает итератор, как показано в листинге 13.13.

Листинг 13.13. Метод insert в vector

```
TEST_CASE("std::vector insert places new elements") {
  std::vector<int> zeros(3, 0); ❶
  auto third_element = zeros.begin() + 2; ❷
  zeros.insert(third_element, 10); ❸
  REQUIRE(zeros[2] == 10); ❹
  REQUIRE(zeros.size() == 4); ❺
}
```

Вектор инициализируется тремя нулями ❶ и генерирует итератор, указывающий на третий элемент, `zeros` ❷. Затем вставляется значение 10 непосредственно перед третьим элементом, передавая итератор и значение 10 ❸. Третий элемент `zeros` теперь равен 10 ❹. Вектор `zeros` содержит четыре элемента ❺.

Каждый раз, когда вы используете `insert`, существующие итераторы становятся недействительными. Например, в листинге 13.13 не нужно повторно использовать

third_element: вектор мог бы изменить свой размер и переместить элемент в памяти, оставив старый итератор зависать в мусорной памяти.

Чтобы вставить элемент в конец vector, используется метод push_back. В отличие от insert, push_back не требует аргумента итератора. Просто предоставляется элемент для копирования в vector, как показано в листинге 13.14.

Листинг 13.14. Метод push_back для vector

```
TEST_CASE("std::vector push_back places new elements") {
  std::vector<int> zeros(3, 0); ❶
  zeros.push_back(10); ❷
  REQUIRE(zeros[3] == 10); ❸
}
```

Опять же vector инициализируется тремя нулями ❶, но на этот раз элемент 10 вставляется в конец vector, используя метод push_back ❷. vector теперь содержит четыре элемента, последний из которых равен 10 ❸.

Можно создавать новые элементы на месте, используя методы emplace и emplace_back. Метод emplace — это вариативный шаблон, который, как и insert, принимает итератор в качестве первого аргумента. Остальные аргументы передаются соответствующему конструктору. Метод emplace_back также является вариативным шаблоном, но, как и для push_back, для него не требуется итератор. Он принимает любое количество аргументов и передает их соответствующему конструктору. Листинг 13.15 показывает эти два метода, используя несколько pair в vector.

Листинг 13.15. Методы emplace_back и emplace вектора

```
#include <utility>

TEST_CASE("std::vector emplace methods forward arguments") {
  std::vector<std::pair<int, int>> factors; ❶
  factors.emplace_back(2, 30); ❷
  factors.emplace_back(3, 20); ❸
  factors.emplace_back(4, 15); ❹
  factors.emplace(factors.begin()❺, 1, 60);
  REQUIRE(factors[0].first == 1); ❻
  REQUIRE(factors[0].second == 60); ❼
}
```

Здесь по умолчанию создается vector, содержащий несколько pair типа int ❶. Используя метод emplace_back, три pair помещаются в вектор: 2, 30 ❷; 3, 20 ❸; и 4, 15 ❹. Эти значения передаются непосредственно в конструктор pair, который создается на месте. Далее используется emplace для вставки новой pair в начале вектора, передавая результат factor.begin() в качестве первого аргумента ❺. Это заставляет все элементы вектора сместиться вниз, чтобы освободить место для новой pair (1 ❻, 60 ❼).

Поскольку методы размещения могут создавать элементы на месте, кажется, что они должны быть более эффективными, чем методы вставки. Эта интуиция часто

верна, но по сложным и неубедительным причинам это не всегда так. Как правило, используйте методы размещения. Если вы определили узкое место в производительности, попробуйте также методы вставки. Для более глубокого погружения в вопрос см. правило 42 «Эффективного использования C++» Скотта Мейерса.

ПРИМЕЧАНИЕ

В `std::vector<std::pair<int, int>>` нет ничего особенного. Он идентичен любому другому вектору. Отдельные элементы в этом последовательном контейнере просто оказываются `pair`. Поскольку `pair` имеет конструктор, который принимает два аргумента, один для `first` и один для `second`, `emplace_back` может добавить новый элемент, просто передав два значения, которые он должен записать, во вновь созданный `pair`.

Модель хранения

Хотя элементы `vector` являются непрерывными в памяти, как и `array`, сходство на этом заканчивается. У `vector` динамический размер, поэтому он должен иметь возможность изменять размер. Распределитель `vector` управляет динамической памятью, лежащей в основе `vector`.

Поскольку выделения памяти являются затратными, `vector` будет запрашивать больше памяти, чем нужно для текущего числа элементов. Как только он не сможет добавлять больше элементов, он запросит дополнительную память. Память для `vector` всегда является смежной, поэтому, если в конце существующего вектора недостаточно места, он выделит новую область памяти и переместит все элементы вектора в новую область. Количество элементов, которое содержит `vector`, называется его *размером*, а количество элементов, которое он может теоретически удерживать до изменения размера, называется его *емкостью*. На рис. 13.2 — вектор, содержащий три элемента с дополнительной емкостью для еще трех.

Рис. 13.2. Модель хранения в vector

Как показано на рис. 13.2, `vector` продолжается за своим последним элементом. Емкость определяет, сколько элементов `vector` мог бы содержать в этом пространстве. На этом рисунке размер равен трем, а емкость — шести. Можно представить память в `vector` как зрительный зал: его вместимость может составлять 500, а количество зрителей — всего 250.

Результатом такого дизайна является то, что вставка в конец `vector` выполняется очень быстро (если только вектор не нуждается в изменении размера). Вставка

в другое место сопряжена с дополнительными затратами, поскольку вектор должен перемещать элементы, чтобы освободить место.

Можно получить текущую емкость вектора с помощью метода `capacity`, а с помощью метода `max_size` — абсолютную максимальную емкость, на которую `vector` может изменить размер.

Если вы заранее знаете, что потребуется определенная емкость, можно использовать метод `reserve`, который принимает один аргумент `size_t`, соответствующий количеству элементов, для которого нужно задать емкость. С другой стороны, если вы только что удалили несколько элементов и хотите вернуть память распределителю, можно использовать метод `shrink_to_fit`, который объявляет о наличии избыточной емкости. Распределитель может решить уменьшить емкость или нет (это необязательный вызов).

Кроме того, можно удалить все элементы в векторе и установить его размер равным нулю, используя метод `clear`.

В листинге 13.16 показаны все эти связанные с хранилищем методы в единой истории: создается пустой вектор, резервируется место, добавляются некоторые элементы, освобождается избыточная емкость и, наконец, вектор очищается.

Листинг 13.16. Функции управления хранилищем в vector. (Строго говоря, kb_store.capacity()>= 3 ❻❽ не гарантируется, поскольку вызов не является обязательным.)

```
#include <cstdint>
#include <array>

TEST_CASE("std::vector exposes size management methods") {
  std::vector<std::array<uint8_t, 1024>> kb_store; ❶
  REQUIRE(kb_store.max_size() > 0);
  REQUIRE(kb_store.empty()); ❷

  size_t elements{ 1024 };
  kb_store.reserve(elements); ❸
  REQUIRE(kb_store.empty());
  REQUIRE(kb_store.capacity() == elements); ❹

  kb_store.emplace_back();
  kb_store.emplace_back();
  kb_store.emplace_back();
  REQUIRE(kb_store.size() == 3); ❺

  kb_store.shrink_to_fit();
  REQUIRE(kb_store.capacity() >= 3); ❻

  kb_store.clear(); ❼
  REQUIRE(kb_store.empty());
  REQUIRE(kb_store.capacity() >= 3); ❽
}
```

Создается `vector` объектов `array` с названием `kb_store`, в котором хранятся блоки по 1 Кб ❶. Если не использовать специальную платформу без динамической памяти, `kb_store.max_size()` будет больше нуля, потому при инициализации `vector` по умолчанию является пустым ❷.

Затем резервируется 1024 элемента ❸, что не меняет статус пустого вектора, но увеличивает его емкость, чтобы соответствовать ❹. `vector` теперь имеет 1024 × 1 Кб = = 1 Мб непрерывного зарезервированного пространства. Зарезервировав место, вы добавляете три массива и проверяете, что `kb_store.size()` увеличилась соответственно ❺.

Вы зарезервировали место для 1024 элементов. Чтобы освободить 1024 – 3 = 1021 неиспользуемый элемент, вызывается `shrink_to_fit`, который уменьшает емкость до 3 ❻.

Наконец, вызывается `clear` для `vector` ❼, который разрушает все элементы и уменьшает его размер до нуля. Однако емкость остается неизменной, поскольку еще один вызов `shrink_to_fit` не был выполнен ❽. Это важно, потому что вектор не хочет выполнять дополнительную работу, если вы собираетесь снова добавлять элементы.

Неполный список поддерживаемых операций

В таблице 13.2 приведен неполный список операций над `vector`. В этой таблице `v`, `v1` и `v2` имеют тип `std::vector<T>`, `t` имеет тип `T`, `alc` означает соответствующий распределитель, а `itr` — итератор. Звездочка (`*`) указывает, что эта операция делает недействительными обычные указатели и итераторы для элементов `v`, по крайней мере при некоторых обстоятельствах.

Таблица 13.2. Неполный список операций `std::vector`

Операция	Примечания
`vector<T>{..., [alc]}`	Выполняет фигурную инициализацию только что построенного вектора. По умолчанию используется `alc = std::allocator<T>`
`vector<T>(s, [t], [alc])`	Заполняет только что построенный вектор числом `s` копий `t`. Если `t` не указано, по умолчанию создаются экземпляры `T`
`vector<T>(v) D`	Глубоко копирует `v`; выделяет новую память
`vector<T>(move(v))`	Принимает владение памятью, элементами в `v`. Нет выделения памяти
`~vector`	Уничтожает все элементы, содержащиеся в векторе, и освобождает динамическую память
`v.begin()`	Возвращает итератор, указывающий на первый элемент
`v.cbegin()`	Возвращает `const`-итератор, указывающий на первый элемент
`v.end()`	Возвращает итератор, указывающий на следующий после последнего элемента

Операция	Примечания
`v.cend()`	Возвращает `const`-итератор, указывающий на следующий после последнего элемента
`v1 = v2`	`v1` уничтожает свои элементы; каждый элемент `v2` копируется. Распределяет память, только если ему нужно изменить размер, чтобы соответствовать элементам `v2` *
`v1 = move(v2)`	`v1` уничтожает свои элементы; каждый элемент `v2` перемещается. Распределяет память, только если ему нужно изменить размер, чтобы соответствовать элементам `v2` *
`v.at(0)`	Получает доступ к элементу `0` из `v`. Выдает `std::out_of_range`, если выходит за пределы вектора
`v[0]`	Получает доступ к элементу 0 из `v`. Выдает неопределенное поведение, если выходит за пределы вектора
`v.front()`	Получает доступ к первому элементу
`v.back()`	Получает доступ к последнему элементу
`v.data()`	Возвращает обычный указатель на первый элемент, если массив не пустой. Для пустых массивов возвращает действительный, но не разыменованный указатель
`v.assign({... })`	Заменяет содержимое `v` на элементы .... *
`v.assign(s, t)`	Заменяет содержимое `v` на количество `s` копий `t` *
`v.empty()`	Возвращает `true`, если размер вектора равен нулю; иначе — `false`
`v.size()`	Возвращает количество элементов в векторе
`v.capacity()`	Возвращает максимальное количество элементов, которое вектор может содержать без изменения размера
`v.shrink_to_fit()`	Может уменьшить объем памяти вектора так, чтобы `capacity()` равнялась `size()` *
`v.resize(s, [t])`	Изменяет размер `v`, чтобы содержать `s` элементов. Если это уменьшает `v`, уничтожает элементы в конце. Если это увеличивает `v`, вставляет созданный по умолчанию `Ts` или копии `t`, если они предоставлены *
`v.reserve(s)`	Увеличивает память вектора, чтобы он мог содержать как минимум `s` элементов *
`v.max_size()`	Возвращает максимально возможный размер, на который вектор может изменить размер
`v.clear()`	Удаляет все элементы в `v`, но оставляет емкость *
`v.insert(itr, t)`	Вставляет копию `t` непосредственно перед элементом, на который указывает `itr`; диапазон `v` должен содержать `itr` *
`v.push_back(t)`	Вставляет копию `t` в конце `v` *

Продолжение ⇩

Таблица 13.2 *(продолжение)*

Операция	Примечания
`v.emplace(itr,...)`	Создает `T` на месте, передавая аргументы... соответствующему конструктору. Элемент вставляется непосредственно перед элементом, на который указывает `itr` *
`v.emplace_back(...)`	Создает `T` на месте, передавая аргументы... соответствующему конструктору. Элемент вставляется в конце `v` *
`v1.swap(v2)` `swap(v1, v2)`	Заменяет все элементы `v1` на элементы `v2` *

`v1 == v2`
`v1 != v2`
`v1 > v2`
`v1 >= v2`
`v1 < v2`
`v1 <= v2`
	Равенство верно, если все элементы равны. Сравнение больше/меньше проводится от первого элемента до последнего

Узкоспециализированные контейнеры последовательности

Контейнеры `vector` и `array` являются очевидным выбором в большинстве ситуаций, в которых нужна последовательная структура данных. Если вы знаете заранее необходимое количество элементов, используйте `array`. Если вы этого не сделаете, используйте `vector`.

Можно оказаться в особой ситуации, когда вектор и массив не имеют требуемых характеристик производительности. В этом разделе освещается ряд альтернативных последовательных контейнеров, которые могут предложить превосходные характеристики производительности в такой ситуации.

Двусторонняя очередь

Двусторонняя очередь (*deque*) представляет собой последовательный контейнер с быстрой вставкой и удалением операций спереди и сзади. Реализация STL `std::deque` доступна из заголовка `<deque>`.

ПРИМЕЧАНИЕ

Библиотека Boost Container также содержит `boost::container::deque` в заголовке `<boost/container/deque.hpp>`.

`vector` и `deque` имеют очень похожие интерфейсы, но внутренне их модели хранения совершенно разные: `vector` гарантирует, что все элементы последовательны в памяти, тогда как память `deque` обычно разбросана, как гибрид между `vector` и `list`. Это

делает большие операции по изменению размера более эффективными и позволяет быстро вставлять/удалять элементы в передней части контейнера.

Создание и доступ к членам — идентичные операции для `vector` и `deque`.

Поскольку внутренняя структура `deque` сложна, она не предоставляет метод `data`. В обмен вы получаете доступ к `push_front` и `emplace_front`, которые отражают `push_back` и `emplace_back`, с которыми вы знакомы по `vector`. В листинге 13.17 показано, как использовать `push_back` и `push_front` для вставки значений в набор символов.

Листинг 13.17. deque поддерживает push_front и push_back

```
#include <deque>

TEST_CASE("std::deque supports front insertion") {
  std::deque<char> deckard;
  deckard.push_front('a'); ❶ // a
  deckard.push_back('i'); ❷ // ai
  deckard.push_front('c'); // cai
  deckard.push_back('n');  // cain
  REQUIRE(deckard[0] == 'c'); ❸
  REQUIRE(deckard[1] == 'a');
  REQUIRE(deckard[2] == 'i');
  REQUIRE(deckard[3] == 'n');
}
```

После создания пустой `deque` чередующиеся буквы передаются в начало ❶ и конец ❷ `deque`, чтобы она содержала элементы `c`, `a`, `i` и `n` ❸.

ПРИМЕЧАНИЕ

Плохой идеей будет пытаться извлечь здесь строку, например `&deckard[0]`, потому что `deque` не дает никаких гарантий относительно внутреннего размещения.

Ниже перечислены методы `vector`, не реализованные `deque`, и объясняется их отсутствие:

- `capacity`, `reserve`

  Поскольку внутренняя структура сложна, вычисление емкости может быть неэффективным. Кроме того, выделения памяти в `deque` относительно быстрые, потому что `deque` не перемещает существующие элементы, поэтому резервирование памяти заранее не требуется.

- `data`

  Элементы `deque` не являются непрерывными.

Таблица 13.3 отражает дополнительные операторы, предлагаемые `deque`, но не `vector`. В этой таблице `d` имеет тип `std::deque<T>`, а `t` имеет тип `T`. Звездочка (*) указывает, что эта операция делает недействительными итераторы для элементов `v`, по крайней мере в некоторых случаях. (Указатели на существующие элементы остаются действительными.)

Таблица 13.3. Неполный список операций `std::deque`

Операция	Примечания
`d.emplace_front(...)`	Создает элемент на месте перед `d`, передавая все аргументы в соответствующий конструктор *
`d.push_front(t)`	Создает элемент на месте перед `d`, копируя `t` *
`d.pop_front()`	Удаляет элемент в начале `d` *

Список

Список (`list`) — это контейнер последовательности с операциями быстрой вставки/удаления в любом месте, но без случайного доступа к элементу. Реализация STL `std::list` доступна из заголовка `<list>`.

ПРИМЕЧАНИЕ

Библиотека Boost Container также содержит `boost::container::list` в заголовке `<boost/container/list.hpp>`.

Список реализован в виде двусвязного списка, структуры данных, состоящей из *узлов*. Каждый узел содержит элемент, прямую ссылку и обратную ссылку. Он полностью отличается от `vector`, который хранит элементы в смежной памяти. В результате нельзя использовать `operator[]` или `at` для доступа к произвольным элементам в списке, потому что такие операции будут очень неэффективными. (Эти методы просто недоступны в `list` из-за их ужасных характеристик производительности.) Компромисс в том, что вставка и удаление элементов в списке происходит намного быстрее. Все, что нужно обновить, — это прямые и обратные ссылки соседей элемента вместо тасования потенциально больших, непрерывных диапазонов элементов.

Контейнер `list` поддерживает те же шаблоны конструктора, что и `vector`.

Можно выполнять специальные операции со списками, такие как объединение элементов из одного списка в другой, используя метод `splice`, удаление последовательно повторяющихся элементов с использованием метода `unique` и даже сортировку элементов контейнера с помощью метода `sort`. Рассмотрим, например, метод `remove_if`. Метод `remove_if` принимает объект функции в качестве параметра и просматривает `list`, вызывая объект функции для каждого элемента. Если результат равен `true`, `remove_if` удаляет элемент. В листинге 13.18 показано, как использовать метод `remove_if` для удаления всех четных чисел в `list` с помощью лямбда-предиката.

Здесь используется фигурная инициализация для заполнения `list` объектов `int` ❶. Затем используется метод `remove_if` для удаления всех четных чисел ❷. Поскольку только четные числа по модулю 2 равны нулю, эта лямбда-функция проверяет, является ли число четным. Чтобы проверить, что `remove_if` извлек четные элемен-

ты 22 и 44, создается итератор, указывающий на начало списка ❸, проверяется его значение ❹ и увеличивается ❺, пока не будет достигнут конец списка ❻.

Листинг 13.18. Список поддерживает remove_if

```
#include <list>

TEST_CASE("std::list supports front insertion") {
  std::list<int> odds{ 11, 22, 33, 44, 55 }; ❶
  odds.remove_if([](int x) { return x % 2 == 0; }); ❷
  auto odds_iter = odds.begin(); ❸
  REQUIRE(*odds_iter == 11); ❹
  ++odds_iter; ❺
  REQUIRE(*odds_iter == 33);
  ++odds_iter;
  REQUIRE(*odds_iter == 55);
  ++odds_iter;
  REQUIRE(odds_iter == odds.end()); ❻
}
```

Все методы `vector`, не реализованные `list`, вместе с объяснением их отсутствия:

- **`capacity, reserve, shrink_to_fit.`** Поскольку список получает память `list`, для него не требуется периодическое изменение размера;
- **`operator[].`** Случайный доступ к элементам в списках слишком затратен;
- **`data.`** Не нужен, потому что элементы списка не являются непрерывными.

Таблица 13.4 обобщает дополнительные операторы, предлагаемые `list`, но не `vector`. В этой таблице `lst`, `lst1` и `lst2` имеют тип `std::list<T>`, а `t` имеет тип `T`. Аргументы `itr1`, `itr2a` и `itr2b` являются итераторами списка. Звездочка (*) указывает, что операция делает недействительными итераторы для элементов `v`, по крайней мере в некоторых случаях. (Указатели на существующие элементы остаются действительными.)

Таблица 13.4. Неполный список операций `std::list`

Операция	Примечания
`lst.emplace_front(...)`	Создает элемент на месте перед `d`, перенаправляя все аргументы в соответствующий конструктор
`lst.push_front(t)`	Создает элемент на месте перед `d`, копируя `t`
`lst.pop_front()`	Удаляет элемент в передней части `d`
`lst.push_back(t)`	Создает элемент на месте в конце `d`, копируя `t`
`lst.pop_back()`	Удаляет элемент в конце `d`
`lst1.splice(itr1, lst2, [itr2a], [itr2b])`	Переносит элементы из `lst2` в `lst1` на позицию `itr1`. Альтернативно переносит только элемент в `itr2a` или элементы в полуоткрытом диапазоне `itr2a` в `itr2b`

Продолжение ⇩

Таблица 13.4 *(продолжение)*

Операция	Примечания
`lst.remove(t)`	Удаляет все элементы в списке, равном `t`
`lst.remove_if(pred)`	Исключает элементы в `lst`, где `pred` возвращает `true`; `pred` принимает один аргумент `T`
`lst.unique(pred)`	Удаляет дубликаты последовательных элементов в `lst` согласно объекту функции `pred`, который принимает два аргумента `T` и возвращает `t1 == t2`
`lst1.merge(lst2, comp)`	Объединяет `lst1` и `lst2` в соответствии с объектом функции `comp`, который принимает два аргумента `T` и возвращает `t1 < t2`
`lst.sort(comp)`	Сортирует `lst` по объекту функции `comp`
`lst.reverse()`	Меняет порядок элементов `lst` (изменяет `lst`)

ПРИМЕЧАНИЕ

`STL` также предлагает `std::forward_list` в заголовке `<forward_list>`, который является односвязным списком и допускает итерацию только в одном направлении. `forward_list` немного более эффективен, чем `list`, и он оптимизирован для ситуаций, в которых нужно хранить очень мало (или вообще ни одного) элементов.

Стеки

STL предоставляет три *контейнера-адаптера*, которые инкапсулируют другие контейнеры STL и предоставляют специальные интерфейсы для специализированных ситуаций. Адаптеры — это стек, очередь и очередь приоритетов.

Стек — это структура данных с двумя основными операциями: `push` и `pop`. При *помещении* элемента в стек элемент вставляется в конец стека. При *извлечении* элемента из стека элемент удаляется из конца стека. Это расположение называется «*последним пришел — первым вышел*»: последний элемент, который должен быть помещен в стек, является первым, который извлекается.

STL предлагает `std::stack` в заголовке `<stack>`. Шаблон класса `stack` принимает два параметра шаблона. Первый — это базовый тип упакованного контейнера, например `int`, а второй — тип упакованного контейнера, например `deque` или `vector`. Этот второй аргумент является необязательным и по умолчанию имеет значение `deque`.

Чтобы создать стек, можно передать ссылку на `deque`, `vector` или `list` для инкапсуляции. Таким образом, стек преобразует свои операции, такие как `push` и `pop`, в методы, которые понимает базовый контейнер, такие как `push_back` и `pop_back`. Если не указывать аргумент конструктора, по умолчанию в стеке используется `deque`. Второй параметр шаблона должен соответствовать типу этого контейнера.

Чтобы получить ссылку на элемент вверху стека, используется метод `top`.

В листинге 13.19 показано, как использовать `stack` для обертки `vector`.

Листинг 13.19. Использование stack для обертки vector

```
#include <stack>

TEST_CASE("std::stack supports push/pop/top operations") {
  std::vector<int> vec{ 1, 3 }; ❶ // 1 3
  std::stack<int, decltype(vec)> easy_as(vec); ❷
  REQUIRE(easy_as.top() == 3); ❸
  easy_as.pop();  ❹                  // 1
  easy_as.push(2); ❺                 // 1 2
  REQUIRE(easy_as.top() == 2); ❻
  easy_as.pop();                 // 1
  REQUIRE(easy_as.top() == 1);
  easy_as.pop();                 //
  REQUIRE(easy_as.empty()); ❼
}
```

Создается `vector` с типами `int` и названием `vec`, содержащий элементы 1 и 3 ❶. Затем `vec` передается в конструктор нового `stack`, предоставляя второй параметр шаблона `decltype(vec)` ❷. Теперь верхний элемент в стеке равен 3, потому что это последний элемент в `vec` ❸. После первого вызова `pop` ❹ новый элемент 2 помещается в `stack` ❺. Теперь верхний элемент равен 2 ❻. После очередной серии `pop-top-pop` `stack` становится пустым ❼.

Таблица 13.5 обобщает операции со `stack`. В этой таблице `s`, `s1` и `s2` имеют тип `std::stack<T>`; `t` имеет тип `T`; и `ctr` обозначает контейнер типа `ctr_type<T>`.

Таблица 13.5. Список операций `std::stack`

Операция	Примечания
`stack<T, [ctr_type <T>]>([ctr])`	Создает стек с типами `T`, используя `ctr` в качестве внутренней ссылки на контейнер. Если контейнер не предоставлен, создает пустую `deque`
`s.empty()`	Возвращает `true`, если контейнер пуст
`s.size()`	Возвращает количество элементов в контейнере
`s.top()`	Возвращает ссылку на элемент в верхней части стека
`s.push(t)`	Помещает копию `t` в конец контейнера
`s.emplace(...)`	Создает `T` на месте, перенаправляя ... соответствующему конструктору
`s.pop()`	Удаляет элемент в конце контейнера
`s1.swap(s2)` `swap(s1, s2)`	Меняет содержимое `s2` на `s1`

Очереди

Очередь — это структура данных, которая, как и стек, имеет в качестве основных операций `push` и `pop`. В отличие от стека, очередь работает по принципу «*первым пришел — первым вышел*». При помещении элемента в очередь он вставляется в конец. При извлечении элемента из очереди он удаляется из начала. Таким образом, элемент, который был в очереди дольше всех, удаляется первым.

STL предлагает `std::queue` в заголовке `<queue>`. Как и `stack`, `queue` принимает два параметра шаблона. Первый параметр — это базовый тип упакованного контейнера, а необязательный второй параметр — это тип упакованного контейнера, который по умолчанию также имеет значение `deque`.

Среди контейнеров STL можно использовать `deque` или `list` в качестве основного контейнера для `queue`, потому что добавление и удаление элементов в `vector` неэффективно.

Можно получить доступ к элементу в начале или в конце очереди, используя методы `front` и `back`.

Листинг 13.20 показывает, как использовать очередь, чтобы обернуть `deque`.

Листинг 13.20. Использование queue для обертки deque

```
#include <queue>

TEST_CASE("std::queue supports push/pop/front/back") {
  std::deque<int> deq{ 1, 2 }; ❶
  std::queue<int> easy_as(deq); ❷   // 1 2

  REQUIRE(easy_as.front() == 1); ❸
  REQUIRE(easy_as.back() == 2); ❹
  easy_as.pop(); ❺                  // 2
  easy_as.push(3); ❻                // 2 3
  REQUIRE(easy_as.front() == 2); ❼
  REQUIRE(easy_as.back() == 3); ❽
  easy_as.pop();                   // 3
  REQUIRE(easy_as.front() == 3);
  easy_as.pop();                   //
  REQUIRE(easy_as.empty()); ❾
}
```

Код начинается с `deque`, содержащей элементы 1 и 2 ❶, и которая передается в очередь с названием `easy_as` ❷. Используя методы `front` и `back`, можно проверить, что очередь начинается с 1 ❸ и заканчивается 2 ❹. После вставки первого элемента, 1, остается очередь, содержащая только один элемент 2 ❺. Затем в конец очереди добавляется 3 ❻, поэтому метод `front` дает 2 ❼, а `back` возвращает 3 ❽. После еще двух итераций `pop-front` останется пустая `queue` ❾.

Таблица 13.6 обобщает операции с `queue`. В этой таблице `q`, `q1` и `q2` имеют тип `std::queue<T>`; `t` имеет тип `T`; `ctr` означает контейнер типа `ctr_type<T>`.

Таблица 13.6. Список операций `std::queue`

Операция	Примечания
`queue<T, [ctr_type<T>]> ([ctr])`	Создает `queue` с типами `T`, используя `ctr` в качестве внутренней ссылки на контейнер. Если контейнер не предоставлен, создает пустую `deque`
`q.empty()`	Возвращает `true`, если контейнер пуст
`q.size()`	Возвращает количество элементов в контейнере
`q.front()`	Возвращает ссылку на элемент в начале `queue`
`q.back()`	Возвращает ссылку на элемент в конце `queue`
`q.push(t)`	Помещает копию `t` в конец контейнера
`q.emplace(...)`	Создает `T` на месте, перенаправляя... соответствующему конструктору
`q.pop()`	Удаляет элемент в конце контейнера
`q1.swap(q2)` `swap(q1, q2)`	Меняет содержимое `q2` на `q1`

Очереди по приоритету (кучи)

Очередь по приоритету (также называемая кучей) — это структура данных, которая поддерживает операции `push` и `pop` и сохраняет элементы отсортированными в соответствии с пользовательским *объектом сравнения*. Объект сравнения является функциональным объектом, который вызывается с двумя параметрами и возвращает `true`, если первый аргумент меньше второго. При извлечении элементов из очереди по приоритету удаляется самый большой элемент в соответствии с объектом сравнения.

STL предлагает `std::priority_queue` в заголовке `<queue>`. `priority_queue` имеет три параметра шаблона:

- базовый тип упакованного контейнера;
- тип упакованного контейнера;
- тип объекта сравнения.

Только базовый тип является обязательным. Тип упакованного контейнера по умолчанию равен `vector` (возможно, потому что это наиболее широко используемый последовательный контейнер), а тип объекта компаратора по умолчанию равен `std::less`.

ПРИМЕЧАНИЕ

Шаблон класса `std::less` доступен в заголовке `<functions>` и возвращает `true`, если первый аргумент меньше второго.

priority_queue имеет интерфейс, идентичный stack. Единственное отличие состоит в том, что элементы стека удаляются в соответствии с порядком «последним пришел — первым вышел», тогда как очереди по приоритету удаляют элементы в соответствии с критериями объекта сравнения.

Листинг 13.21 показывает основное использование priority_queue.

Листинг 13.21. Базовое использование priority_queue

```
#include <queue>

TEST_CASE("std::priority_queue supports push/pop") {
  std::priority_queue<double> prique; ❶
  prique.push(1.0); // 1.0
  prique.push(2.0); // 2.0 1.0
  prique.push(1.5); // 2.0 1.5 1.0
  REQUIRE(prique.top() == Approx(2.0)); ❷
  prique.pop();     // 1.5 1.0
  prique.push(1.0); // 1.5 1.0 1.0
  REQUIRE(prique.top() == Approx(1.5)); ❸
  prique.pop();     // 1.0 1.0
  REQUIRE(prique.top() == Approx(1.0)); ❹
  prique.pop();     // 1.0
  REQUIRE(prique.top() == Approx(1.0)); ❺
  prique.pop();     //
  REQUIRE(prique.empty()); ❻
}
```

Здесь по умолчанию создается priority_queue ❶, которая инициализирует пустой vector под капотом для хранения его элементов. Элементы 1.0, 2.0 и 1.5 помещаются в priority_queue, которая сортирует элементы в порядке убывания, поэтому контейнер представляет их в порядке 2.0 1.5 1.0.

Выполняется проверка, что top дает 2.0 ❷. Этот элемент извлекается из priority_queue, а затем вызывается push с новым элементом 1.0. Контейнер теперь представляет их в порядке 1.5 ❸ 1.0 ❹ 1.0 ❺, что проверяется серией операций top-pop до тех пор, пока контейнер не станет пустым ❻.

ПРИМЕЧАНИЕ

priority_queue хранит свои элементы в древовидной структуре, поэтому, если заглянуть в ее упакованный контейнер, порядок в памяти не будет соответствовать порядку, подразумеваемому листингом 13.21.

Таблица 13.7 обобщает операции с priority_queue. В этой таблице pq, pq1 и pq2 имеют тип std::priority_queue<T>; t имеет тип T; ctr — это контейнер типа ctr_type <T>; srt означает контейнер типа srt_type<T>.

Таблица 13.7. Список операций `std::priority_queue`

Операция	Примечания
`priority_queue <T, [ctr_type <T>], [cmp_type] >([cmp], [ctr])`	Создает `priority_queue` из типов `T`, используя `ctr` в качестве внутреннего контейнера и `cmp` в качестве объекта сравнения. Если контейнер не предоставлен, создает пустую `deque`. Использует `std::less` как сортировщик по умолчанию
`pq.empty()`	Возвращает `true`, если контейнер пуст
`pq.size()`	Возвращает количество элементов в контейнере
`pq.top()`	Возвращает ссылку на самый большой элемент в контейнере
`pq.push(t)`	Помещает копию `t` в конец контейнера
`pq.emplace(...)`	Создает `T` на месте, перенаправляя ... соответствующему конструктору
`pq.pop()`	Удаляет элемент в конце контейнера
`pq1.swap(pq2)` `swap(pq1, pq2)`	Меняет содержимое `s2` на `s1`

Наборы битов

Набор битов (bitset) — это структура данных, в которой хранится битовая последовательность фиксированного размера. Есть возможность управлять каждым битом.

STL предлагает `std::bitset` в заголовке `<bitset>`. Шаблон класса `bitset` принимает один параметр шаблона, соответствующий желаемому размеру. Можно достичь аналогичной функциональности, используя массив `bool`, но набор битов оптимизирован для экономии пространства и обеспечивает некоторые специальные удобные операции.

ПРИМЕЧАНИЕ

`STL` специализируется на `std::vector <bool>`, поэтому может выиграть от той же эффективности использования пространства, что и набор битов. (Вспомните из раздела «Специализация шаблона» на с. 242, что специализация шаблона — это процесс повышения эффективности определенных видов шаблонов.) Boost предлагает `boost::dynamic_bitset`, который обеспечивает динамическое изменение размеров во время выполнения.

Созданный по умолчанию `bitset` содержит все нулевые (ложные) биты. Чтобы инициализировать `bitset` с другим содержимым, можно предоставить значение `unsignedlong long`. Побитовое представление его как целого устанавливает значение `bitset`. Можно получить доступ к отдельным битам в `bitset`, используя `operator[]`. В листинге 13.22 показано, как инициализировать `bitset` целочисленным литералом и извлечь его элементы.

Листинг 13.22. Инициализация набора битов целым числом

```
#include <bitset>

TEST_CASE("std::bitset supports integer initialization") {
  std::bitset<4> bs(0b1010); ❶
  REQUIRE_FALSE(bs[0]); ❷
  REQUIRE(bs[1]); ❸
  REQUIRE_FALSE(bs[2]); ❹
  REQUIRE(bs[3]); ❺
}
```

`bitset` инициализируется с помощью 4-битного *полубайта* `0101` ❶. Итак, первый элемент ❷ и третий элемент ❹ равны нулю, а второй элемент ❸ и четвертый элемент ❺ равны 1.

Также можно предоставить строковое представление желаемого `bitset`, как показано в листинге 13.23.

Листинг 13.23. Инициализация набора битов строкой

```
TEST_CASE("std::bitset supports string initialization") {
  std::bitset<4> bs1(0b0110); ❶
  std::bitset<4> bs2("0110"); ❷
  REQUIRE(bs1 == bs2); ❸
}
```

Здесь вы создаете `bitset` с названием `bs1`, используя тот же самый целочисленный полубайт `0b0110` ❶, и другой `bitset` с названием `bs2`, используя строковый литерал 0110 ❷. Оба этих подхода инициализации создают идентичные объекты `bitset` ❸.

Таблица 13.8 обобщает операции с `bitset`. В этой таблице `bs`, `bs 1` и `bs2` имеют тип `std::bitset<N>`, а `i` означает `size_t`.

Таблица 13.8. Список операций `std::bitset`

Операция	Примечания
`bitset<N>([val])`	Создает `bitset` с начальным значением `val`, которое может быть либо строкой 0 и 1, либо `unsigned long long`. Конструктор по умолчанию инициализирует все биты нулями
`bs[i]`	Возвращает значение `i`-го бита: 1 возвращает истину, 0 — ложь
`bs.test(i)`	Возвращает значение `i`-го бита: 1 возвращает истину, 0 — ложь. Выполняет проверку границ; выбрасывает `std::out_of_range`
`bs.set()`	Устанавливает значение 1 для всех битов
`bs.set(i, val)`	Устанавливает значение `val` для `i`-го бита. Выполняет проверку границ; выбрасывает `std::out_of_range`
`bs.reset()`	Устанавливает значение 0 для всех битов

Операция	Примечания
`bs.reset(i)`	Устанавливает значение 0 для `i`-го бита. Выполняет проверку границ; выбрасывает `std::out_of_range`
`bs.flip()`	Меняет все биты: (0 становится 1; 1 становится 0)
`bs.flip(i)`	Меняет значение `i`-го бита на 0. Выполняет проверку границ; выбрасывает `std::out_of_range`
`bs.count()`	Возвращает количество битов, равных 1
`bs.size()`	Возвращает размер `N` набора битов
`bs.any()`	Возвращает `true`, если хотя бы один из битов равен 1
`bs.none()`	Возвращает `true`, если все биты равны 0
`bs.all()`	Возвращает `true`, если все биты равны 1
`bs.to_string()`	Возвращает `string`-представление `bitset`
`bs.to_ulong()`	Возвращает `unsigned long`-представление `bitset`
`bs.to_ullong()`	Возвращает `unsigned long long`-представление `bitset`

Специальные контейнеры последовательности в Boost

Boost предоставляет множество специальных контейнеров, и здесь просто не хватает места, чтобы изучить все их возможности. В таблице 13.9 приведены имена, заголовки и краткие описания некоторых из них.

ПРИМЕЧАНИЕ

Обратитесь к документации Boost Container для получения дополнительной информации.

Таблица 13.9. Специальные контейнеры Boost

Класс/Заголовок	Описание
`boost::intrusive::*` `<boost/intrusive/*.hpp>`	Навязчивые контейнеры предъявляют требования к элементам, которые они содержат (например, наследование от определенного базового класса). В обмен они предлагают существенный прирост производительности
`boost::container::stable_vector` `<boost/container/stable_vector.hpp>`	Вектор без смежных элементов, но он гарантирует, что итераторы и ссылки на элементы остаются действительными до тех пор, пока элемент не будет удален (как со списком)
`boost::container::slist` `<boost/container/slist.hpp>`	`forward_list`с быстрым методом `size`

Продолжение ⇩

Таблица 13.9 *(продолжение)*

Класс/Заголовок	Описание
`boost::container::static_vector` `<boost/container/static_vector.hpp>`	Гибрид между массивом и вектором, который хранит динамическое число элементов до фиксированного размера. Элементы хранятся в памяти `stable_vector` как `array`
`boost::container::small_vector` `<boost/container/small_vector.hpp>`	Вектороподобный контейнер, оптимизированный для хранения небольшого количества элементов. Содержит некоторое заранее выделенное пространство, избегая динамического выделения
`boost::circular_buffer` `<boost/circular_buffer.hpp>`	Контейнер в виде очереди фиксированной емкости, который заполняет элементы по кругу; новый элемент перезаписывает самый старый элемент при достижении емкости
`boost::multi_array` `<boost/multi_array.hpp>`	Массивоподобный контейнер, который принимает несколько измерений. Вместо того чтобы, например, иметь массив массивов массивов, можно указать трехмерный `multi_array x`, который разрешает доступ к элементу, например `x [5] [1] [2]`
`boost::ptr_vector` `boost::ptr_list` `<boost/ptr_container/*.hpp>`	Наличие коллекции умных указателей может быть неоптимальным. Векторы-указатели управляют коллекцией динамических объектов более эффективным и удобным способом

ПРИМЕЧАНИЕ

Boost Intrusive также содержит некоторые специализированные контейнеры, которые в определенных ситуациях повышают производительность. В первую очередь это полезно для разработчиков библиотек.

Ассоциативные контейнеры

Ассоциативные контейнеры обеспечивают очень быстрый поиск элементов. Последовательные контейнеры имеют некоторое естественное упорядочение, которое позволяет выполнять итерации от начала контейнера до конца в точно указанном порядке. Ассоциативные контейнеры немного отличаются. Это семейство контейнеров разделяется по трем осям:

- содержат ли элементы ключи (множество) или пары ключ-значение (ассоциативный массив);
- упорядочены ли элементы;
- являются ли ключи *уникальными*.

Множества

`std::set`, доступный в заголовке `<set>` в STL, является ассоциативным контейнером, который содержит отсортированные уникальные элементы, называемые ключами. Поскольку `set` хранит отсортированные элементы, можно эффективно вставлять, удалять и искать элементы. Кроме того, `set` поддерживает отсортированную итерацию по его элементам, и можно полностью контролировать, как ключи сортируются с использованием объектов сравнения.

ПРИМЕЧАНИЕ

Boost также предоставляет `boost::container::set` в заголовке `<boost/container/set.hpp>`.

Создание

Шаблон класса `set<T, Comparator, Allocator>` принимает три параметра шаблона:

- тип ключа `T`;
- тип оператора сравнения, который по умолчанию равен `std::less`;
- тип распределителя, который по умолчанию равен `std::allocator<T>`.

Множества предоставляют большую гибкость при создании. Каждый из следующих конструкторов принимает необязательные оператор сравнения и распределитель (типы которых должны соответствовать их соответствующим параметрам шаблона):

- конструктор по умолчанию, который инициализирует пустое `set`;
- конструкторы переноса и копирования с обычным поведением;
- конструктор диапазона, который копирует элементы из диапазона в `set`;
- фигурный инициализатор.

В листинге 13.24 показан каждый из этих конструкторов.

Листинг 13.24. Конструкторы множества

```
#include <set>

TEST_CASE("std::set supports") {
  std::set<int> emp; ❶
  std::set<int> fib{ 1, 1, 2, 3, 5 }; ❷
  SECTION("default construction") {
    REQUIRE(emp.empty()); ❸
  }
  SECTION("braced initialization") {
    REQUIRE(fib.size() == 4); ❹
  }
  SECTION("copy construction") {
    auto fib_copy(fib);
    REQUIRE(fib.size() == 4); ❺
    REQUIRE(fib_copy.size() == 4); ❻
```

```
    }
    SECTION("move construction") {
      auto fib_moved(std::move(fib));
      REQUIRE(fib.empty()); ❼
      REQUIRE(fib_moved.size() == 4); ❽
    }
    SECTION("range construction") {
      std::array<int, 5> fib_array{ 1, 1, 2, 3, 5 };
      std::set<int> fib_set(fib_array.cbegin(), fib_array.cend());
      REQUIRE(fib_set.size() == 4); ❾
    }
  }
```

В коде создаются два разных набора с помощью фигурной инициализации ❷ и конструктора по умолчанию ❶. Созданное по умолчанию множество с именем `emp` является пустым ❸, а множество, инициализированное с фигурными скобками с именем `fib`, имеет четыре элемента ❹. В фигурный инициализатор добавляются пять элементов, почему же в итоге остаются только четыре элемента? Напомним, что элементы множества уникальны, поэтому 1 добавляется только один раз.

Затем создается `fib` с помощью конструктора копирования, в результате чего получается два множества размером 4 ❺ ❻. С другой стороны, конструктор переноса очищает перемещенное множество ❼ и переносит элементы в новое множество ❽.

Затем можно инициализировать множество из диапазона. Создается массив из пяти элементов, а затем он передается как диапазон в конструктор множеств с помощью методов `cbegin` и `cend`. Как и в случае инициализации со скобками ранее в коде, множество содержит только четыре элемента, потому что дубликаты отбрасываются ❾.

Семантика перемещения и копирования

Помимо конструкторов перемещения/копирования, также доступны операторы присваивания переноса/копирования. Как и в случае других операций копирования контейнера, копирование множества может быть очень медленным, поскольку каждый элемент должен копироваться, а операции переноса обычно выполняются быстро, поскольку элементы находятся в динамической памяти. Можно просто передать право владения множеством, не нарушая элементы.

Доступ к элементам

Существует несколько вариантов извлечения элементов из множества. Основной метод — это `find`, который принимает `const` ссылку на ключ и возвращает итератор.

Если множество содержит ключ, соответствующий элементу, `find` возвращает итератор, указывающий на найденный элемент. Если это не так, он вернет итератор, указывающий на `end`. Метод `lower_bound` возвращает итератор к первому элементу, который *не меньше, чем* аргумент ключа, тогда как метод `upper_bound` возвращает первый элемент, *больший, чем* заданный ключ.

Класс `set` поддерживает два дополнительных метода поиска, в основном для совместимости неуникальных ассоциативных контейнеров.

- Метод `count` возвращает количество элементов, соответствующих ключу. Поскольку элементы множества являются уникальными, `count` возвращает 0 или 1.
- Метод `equal_range` возвращает полуоткрытый диапазон, содержащий все элементы, соответствующие данному ключу. Диапазон возвращает `std::pair` итераторов с `first`, указывающим на соответствующий элемент, и `second`, указывающим на элемент после `first`. Если `equal_range` не находит соответствующего элемента, `first` и `second` указывают на первый элемент, который больше указанного ключа. Другими словами, пара, возвращаемая `equal_range`, эквивалентна паре `lower_bound` в качестве `first` и `upper_bound` в качестве `second`.

Листинг 13.25 показывает эти два метода доступа.

Листинг 13.25. Доступ к элементам set

```
TEST_CASE("std::set allows access") {
  std::set<int> fib{ 1, 1, 2, 3, 5 }; ❶
  SECTION("with find") { ❷
    REQUIRE(*fib.find(3) == 3);
    REQUIRE(fib.find(100) == fib.end());
  }
  SECTION("with count") { ❸
    REQUIRE(fib.count(3) == 1);
    REQUIRE(fib.count(100) == 0);
  }
  SECTION("with lower_bound") { ❹
    auto itr = fib.lower_bound(3);
    REQUIRE(*itr == 3);
  }
  SECTION("with upper_bound") { ❺
    auto itr = fib.upper_bound(3);
    REQUIRE(*itr == 5);
  }
  SECTION("with equal_range") { ❻
    auto pair_itr = fib.equal_range(3);
    REQUIRE(*pair_itr.first == 3);
    REQUIRE(*pair_itr.second == 5);
  }
}
```

Сначала создается множество из четырех элементов 1 2 3 5 ❶. Используя `find`, можно извлечь итератор для элемента 3. Также можно определить, что 8 не входит во множество, потому что `find` возвращает итератор, указывающий на `end` ❷. Можно определить аналогичную информацию с помощью `count`, который возвращает 1 при предоставлении ключа 3 и 0 при предоставлении ключа 100 ❸. При передаче 3 методу `lower_bound` он возвращает итератор, указывающий на 3, потому что это первый элемент, который не меньше, чем аргумент ❹. При передаче 3 в `upper_bound`, с другой стороны, получается указатель на элемент 5, потому что это первый элемент

больше, чем аргумент ❺. Наконец, при передаче 3 в метод `equal_range` получается `pair` итераторов. Итератор указывает на 3 `first`, а итератор `second` указывает на 5, элемент сразу после 3 ❻.

`set` также предоставляет итераторы через методы `begin` и `end`, поэтому можно использовать циклы `for`, основанные на диапазоне, для перебора множества от наименьшего элемента к наибольшему.

Добавление элементов

Есть три варианта добавления элементов в `set`:

- `insert` для копирования существующего элемента в множество;
- `emplace` для создания нового элемента в множестве;
- `emplace_hint` для создания нового элемента на месте, как `emplace` (потому что добавление элемента требует сортировки). Разница в том, что метод `emplace_hint` принимает итератор в качестве первого аргумента. Этот итератор является отправной точкой поиска (подсказка). Если итератор находится близко к правильной позиции для вновь вставленного элемента, это может обеспечить существенное ускорение.

Листинг 13.26 показывает несколько способов вставки элементов в множество.

Листинг 13.26. Вставка элемента в множество

```
TEST_CASE("std::set allows insertion") {
  std::set<int> fib{ 1, 1, 2, 3, 5 };
  SECTION("with insert") { ❶
    fib.insert(8);
    REQUIRE(fib.find(8) != fib.end());
  }
  SECTION("with emplace") { ❷
    fib.emplace(8);
    REQUIRE(fib.find(8) != fib.end());
  }
  SECTION("with emplace_hint") { ❸
    fib.emplace_hint(fib.end(), 8);
    REQUIRE(fib.find(8) != fib.end());
  }
}
```

`Insert` ❶ и `emplace` ❷ добавляют элемент `8` в `fib`, поэтому при вызове `find` с `8` получается итератор, указывающий на новый элемент. Можно добиться того же результата чуть более эффективно с помощью `emplace_hint` ❸. Поскольку вы заранее знаете, что новый элемент `8` больше, чем все остальные элементы в `set`, можно использовать `end` как подсказку.

При попытке использовать `insert`, `emplace` или `emplace_hint` для ключа, который уже присутствует в множестве, операция не будет иметь никакого эффекта. Все три метода возвращают `std::pair<Iterator, bool>`, где второй элемент указывает, привела ли операция к вставке (`true`) или нет (`false`). Итератор в `first` указывает

либо на недавно вставленный элемент, либо на существующий элемент, который предотвратил вставку.

Удаление элементов

Можно удалить элементы из множества, используя `erase`, который перегружен для принятия ключа, итератора или полуоткрытого диапазона, как показано в листинге 13.27.

Листинг 13.27. Удаление из множества

```
TEST_CASE("std::set allows removal") {
  std::set<int> fib{ 1, 1, 2, 3, 5 };
  SECTION("with erase") { ❶
    fib.erase(3);
    REQUIRE(fib.find(3) == fib.end());
  }
  SECTION("with clear") { ❷
    fib.clear();
    REQUIRE(fib.empty());
  }
}
```

В первом тесте `erase` вызывается с помощью ключа 3, который удаляет соответствующий элемент из набора. При вызове `find` с параметром 3 вы получаете итератор, указывающий на `end`, что говорит о том, что соответствующий элемент не был найден ❶. Во втором тесте вызывается `clear`, который удаляет все элементы из `set` ❷.

Модель хранения

Операции над множествами выполняются быстро, потому что множества обычно реализуются как *красно-черные деревья*. Эти структуры обрабатывают каждый элемент как узел. Каждый узел имеет одного родителя и до двух детей, его левую и правую ноги. Дочерние элементы каждого узла сортируются, поэтому все дочерние элементы слева меньше дочерних элементов справа.

Таким образом, можно выполнять поиск намного быстрее, чем с помощью линейной итерации, если ветви дерева примерно сбалансированы (равны по длине). Красно-черные деревья имеют дополнительные возможности для восстановления баланса ветвей после вставок и удалений.

ПРИМЕЧАНИЕ

Подробнее о красно-черных деревьях см. в книге «Data Structures and Algorithms in C++» Адама Дроздека (Adam Drozdek).

Неполный список поддерживаемых операций

Таблица 13.10 обобщает операции с `set`. Операции `s`, `s1` и `s2` имеют тип `std::set<T, [cmp_type<T>]>`. `T` — это тип элемента/ключа, а `itr`, `beg` и `end` — заданные итераторы. Переменная `t` — это `T`. Кинжал (†) обозначает метод, который возвращает

`std::pair <Iterator, bool>`, где итератор указывает на результирующий элемент, а значение `bool` равно `true`, если метод вставил элемент, и `false`, если элемент уже существовал.

Таблица 13.10. Операции с `std::set`

Операция	Примечания
`set<T>{..., [cmp], [alc] }`	Выполняет скрытую инициализацию только что созданного множества. По умолчанию используются `cmp=std::less<T>` и `alc=std::allocator<T>`
`set<T>{ beg, end, [cmp], [alc] }`	Конструктор диапазона, который копирует элементы из полуоткрытого диапазона от `beg` до `end`. По умолчанию используются `cmp=std::less<T>` и `alc=std::allocator<T>`
`set<T>(s)`	Глубоко копирует `s`; выделяет новую память
`set<T>(move(s))`	Получает владение элементами в `s`. Нет выделений памяти
`~set`	Уничтожает все элементы, содержащиеся в множестве, и освобождает динамическую память
`s1 = s2`	`s1` уничтожает свои элементы; копирует каждый элемент `s2`. Распределяет память, только если ему нужно изменить размер, чтобы соответствовать элементам `s2`
`s1 = move(s2)`	`s1` уничтожает свои элементы; перемещает каждый элемент `s2`. Распределяет память, только если ему нужно изменить размер, чтобы соответствовать элементам `s2`
`s.begin()`	Возвращает итератор, указывающий на первый элемент
`s.cbegin()`	Возвращает итератор `const`, указывающий на первый элемент
`s.end()`	Возвращает итератор, указывающий на следующий элемент после последнего
`s.cend()`	Возвращает итератор `const`, указывающий на следующий элемент после последнего
`s.find(t)`	Возвращает итератор, указывающий на элемент, соответствующий `t` или `s.end()`, если такого элемента не существует
`s.count(t)`	Возвращает 1, если набор содержит `t`; в противном случае — 0
`s.equal_range(t)`	Возвращает пару итераторов, соответствующих полуоткрытому диапазону элементов, соответствующих `t`
`s.lower_bound(t)`	Возвращает итератор, указывающий на первый элемент не менее чем `t` или `s.end()`, если такого элемента не существует

Операция	Примечания
`s.upper_bound(t)`	Возвращает итератор, указывающий на первый элемент не больше чем `t` или `s.end()`, если такого элемента не существует
`s.clear()`	Удаляет все элементы из набора
`s.erase(t)`	Удаляет элемент, равный `t`
`s.erase(itr)`	Удаляет элемент, на который указывает `itr`
`s.erase(beg, end)`	Удаляет все элементы в полуоткрытом диапазоне от `beg` до `end`
`s.insert(t)`	Вставляет копию `t` в множество †
`s.emplace(...)`	Создает `T` на месте, передавая аргументы... †
`s.emplace_hint(itr,...)`	Создает `T` на месте путем передачи аргументов.... Использует `itr` как подсказку, куда вставить новый элемент †
`s.empty()`	Возвращает `true`, если размер множества равен нулю; в противном случае — `false`
`s.size()`	Возвращает количество элементов в множестве
`s.max_size()`	Возвращает максимальное количество элементов в множестве
`s.extract(t)` `s.extract(itr)`	Получает дескриптор узла, которому принадлежит элемент, соответствующий `t` или на который указывает `itr`. (Это единственный способ удалить элемент только для перемещения)
`s1.merge(s2)` `s1.merge(move(s2))`	Добавляет каждый элемент `s2` в `s1`. Если аргумент является r-значением, перемещает элементы в `s1`
`s1.swap(s2)` `swap(s1, s2)`	Заменяет все элементы `s1` на элементы `s2`

Мультимножества

`std::multiset` (мультимножество), доступное в заголовке `<set>` STL, является ассоциативным контейнером, который содержит отсортированные *неуникальные* ключи. `multiset` поддерживает те же операции, что и `set`, но оно будет хранить избыточные элементы. Это имеет важные последствия для двух методов:

- метод `count` может возвращать значения, отличные от 0 или 1. Метод `count` в `multiset` скажет, сколько элементов соответствует данному ключу;
- метод `equal_range` может возвращать полуоткрытые диапазоны, содержащие более одного элемента. Метод `equal_range` в `multiset` вернет диапазон, содержащий все элементы, соответствующие данному ключу.

Возможно, вы захотите использовать `multiset`, а не `set`, если важно хранить несколько элементов с одним и тем же ключом. Например, можно сохранить всех жителей по адресу, рассматривая адрес как ключ, а каждого жильца дома — как элемент. При использовании `set` можно было бы иметь только одного такого жителя.

В листинге 13.28 показано использование `multiset`.

Листинг 13.28. Доступ к элементам multiset

```
TEST_CASE("std::multiset handles non-unique elements") {
  std::multiset<int> fib{ 1, 1, 2, 3, 5 };
  SECTION("as reflected by size") {
    REQUIRE(fib.size() == 5); ❶
  }
  SECTION("and count returns values greater than 1") {
    REQUIRE(fib.count(1) == 2); ❷
  }
  SECTION("and equal_range returns non-trivial ranges") {
    auto [begin, end] = fib.equal_range(1); ❸
    REQUIRE(*begin == 1); ❹
    ++begin;
    REQUIRE(*begin == 1); ❺
    ++begin;
    REQUIRE(begin == end); ❻
  }
}
```

В отличие от `set` из листинга 13.24, `multiset` допускает несколько единиц, поэтому `size` возвращает 5 — количество элементов, которое было указано в инициализаторах в скобках ❶. При подсчете чисел 1 получится 2 ❷. Можно использовать `equal_range` для перебора этих элементов. Используя синтаксис структурированной привязки, вы получите итераторы `begin` и `end` ❸. Вы перебираете две 1 ❹ ❺ и достигаете конца полуоткрытого диапазона ❻.

Каждая операция в таблице 13.10 работает для мультимножества.

ПРИМЕЧАНИЕ

`Boost` также предоставляет `boost::container::multiset` в заголовке `<boost/container/set.hpp>`.

Неупорядоченные множества

`std::unordered_set`, доступный в заголовке `<unordered_set>` в STL, является ассоциативным контейнером, который содержит *несортированные* уникальные ключи. `unordered_set` поддерживает большинство тех же операций, что и `set` и `multiset`, но его модель внутреннего хранения совершенно иная.

ПРИМЕЧАНИЕ

Boost также предоставляет `boost::unordered_set` в заголовке `<boost/unordered_set.hpp>`.

Вместо того чтобы использовать компаратор для сортировки элементов в красно-черное дерево, `unordered_set` обычно реализуется как хеш-таблица. Возможно, вы захотите использовать `unordered_set` в ситуации, когда нет естественного упорядочения ключей и не нужно перебирать коллекцию в таком порядке. Можно обнаружить, что во многих ситуациях можно использовать либо `set`, либо `unordered_set`. Хотя они выглядят довольно схожими, их внутреннее представление принципиально отличается, поэтому они будут иметь разные характеристики производительности. Если производительность является проблемой, изучите, насколько производителен тот или другой вариант, и используйте более уместный.

Модель хранения: хеш-таблицы

Хеш-функция (hasher) — это функция, которая принимает ключ и возвращает уникальное значение `size_t`, называемое хеш-кодом. `unordered_set` организует свои элементы в хеш-таблицу, которая связывает хеш-код с коллекцией из одного или нескольких элементов, называемой *сегментом*. Чтобы найти элемент, `unordered_set` вычисляет свой хеш-код, а затем просматривает соответствующее поле в хеш-таблице.

Если вы никогда раньше не видели хеш-таблицу, эта информация может быть очень полезна, поэтому давайте рассмотрим пример. Представьте, что есть большая группа людей, которую нужно разбить на какие-то разумные группы, чтобы легко найти человека. Можно сгруппировать людей по дню рождения, что даст вам 365 групп (или 366, если считать 29 февраля в високосном году). День рождения похож на хеш-функцию, которая возвращает одно из 365 значений для каждого человека.

Каждое значение образует группу, и у всех людей в одной и той же группе день рождения одинаковый. В этом примере, чтобы найти человека, нужно сначала определить его день рождения, что даст правильный сегмент. Затем нужного человека можно искать в сегменте.

Пока хеш-функция работает быстро и в каждом сегменте не слишком много элементов, `unordered_set` имеет даже более впечатляющую производительность, чем его упорядоченные аналоги: количество содержащихся элементов не увеличивает время вставки, поиска и удаления. Когда два разных ключа имеют одинаковый хеш-код, это называется *коллизией хеша*. Если существует коллизия хеша, это означает, что два ключа будут находиться в одном и том же сегменте. В предыдущем примере у многих людей будет один и тот же день рождения, поэтому будет много коллизий хеша. Чем больше коллизий хеша, тем больше будет блоков и тем больше времени вы будете тратить на поиск в блоке правильного элемента.

Хеш-функция имеет несколько требований:

- принимает ключ и возвращает хеш-код `size_t`;
- не генерирует исключений;
- равные ключи дают равные хеш-коды;
- неравные ключи дают неравные хеш-коды с высокой вероятностью. (Существует небольшая вероятность коллизии хеша.)

STL предоставляет шаблон класса хеш-функции `std::hash<T>` в заголовке `<functional>`, который содержит специализации для фундаментальных типов, типов перечислений, типов указателей, `optional`, `variant`, умных указателей и многого другого. В качестве примера в листинге 13.29 показано, как `std::hash<long>` соответствует критериям эквивалентности.

Листинг 13.29. std::hash<long> возвращает одинаковые хеш-коды для одинаковых ключей и разные хеш-коды для неравных ключей

```
#include <functional>
TEST_CASE("std::hash<long> returns") {
  std::hash<long> hasher; ❶
  auto hash_code_42 = hasher(42); ❷
  SECTION("equal hash codes for equal keys") {
    REQUIRE(hash_code_42 == hasher(42)); ❸
  }
  SECTION("unequal hash codes for unequal keys") {
    REQUIRE(hash_code_42 != hasher(43)); ❹
  }
}
```

В примере создается хеш-функция типа `std::hash<long>` ❶ и используется для вычисления хеш-кода 42, сохраняя результат в `size_thash_code_42` ❷. При повторном вызове `hasher` с 42 вы получите то же значение ❸. При вызове `hasher` с 43 вместо этого вы получите другое значение ❹.

После того как `unordered_set` хеширует ключ, он может получить сегмент. Поскольку сегмент представляет собой список возможных совпадающих элементов, необходим функциональный объект, который определяет равенство между ключом и элементом сегмента. STL предоставляет шаблон класса `std::equal_to<T>` в заголовке `<functions>`, который просто вызывает `operator==` в своих аргументах, как показано в листинге 13.30.

Листинг 13.30. std::equal_to<long> вызывает operator== для аргументов для определения равенства

```
#include <functional>
TEST_CASE("std::equal_to<long> returns") {
  std::equal_to<long> long_equal_to; ❶
  SECTION("true when arguments equal") {
    REQUIRE(long_equal_to(42, 42)); ❷
  }
  SECTION("false when arguments unequal") {
    REQUIRE_FALSE(long_equal_to(42, 43)); ❸
  }
}
```

Здесь инициализируется `equal_to<long>` с названием `long_equal_to` ❶. При вызове `long_equal_to` с равными аргументами он возвращает `true` ❷. При вызове его с различными аргументами возвращается `false` ❸.

ПРИМЕЧАНИЕ

Для краткости в этой главе не рассматривается реализация пользовательских функций хеширования и эквивалентности, которые понадобятся, если вы хотите создавать неупорядоченные контейнеры с заданными пользователем типами ключей. См. главу 7 второго издания «Стандартной библиотеки C++», Николая Джосаттиса[1].

Создание

Шаблон класса `std::unordered_set<T, Hash, KeyEqual, Allocator>` принимает четыре параметра шаблона:

- тип ключа `T`;
- тип хеш-функции `Hash`, который по умолчанию равен `std::hash<T>`;
- тип функции равенства `KeyEqual`, который по умолчанию равен `std::equal_to <T>`;
- тип распределителя `Allocator`, по умолчанию равен `std::allocator<T>`.

`unordered_set` поддерживает конструкторы, эквивалентные конструкторам `set`, с корректировками для различных параметров шаблона (для `set` требуется `Comparator`, тогда как для `unordered_set` требуются `Hash` и `KeyEqual`). Например, можно использовать `unordered_set` в качестве замены для `set` в листинге 13.24, потому что `unordered_set` имеет конструкторы диапазона и конструкторы копирования/переноса и поддерживает инициализацию в фигурных скобках.

Поддерживаемые операции set

`unordered_set` поддерживает все операции над множествами в таблице 13.10, за исключением `lower_bound` и `upper_bound`, потому что `unordered_set` не сортирует свои элементы.

Управление сегментами

Как правило, причина использования `unordered_set` — высокая производительность. К сожалению, такая производительность обходится дорого: объекты `unordered_set` имеют несколько сложную внутреннюю структуру. Есть различные ручки и регуляторы, которые можно использовать для проверки и изменения этой внутренней структуры во время выполнения.

Первая контрольная мера, которую можно применить, состоит в том, чтобы настроить количество сегментов `unordered_set` (то есть количество сегментов, а не количество элементов в конкретном сегменте). Каждый конструктор `unordered_set` принимает `size_tbucket_count` в качестве первого аргумента, который по умолчанию равен некоторому значению, определяемому реализацией. В таблице 13.11 перечислены основные конструкторы `unordered_set`.

[1] *Джосаттис Н. М.* Стандартная библиотека C++: справочное руководство. — М.: Диалектика-Вильямс, 2019. — 1136 с.

Таблица 13.11. Конструкторы `unordered_set`

Операция	Примечания
`unordered_set<T>([bck], [hsh], [keq], [alc])`	Размер контейнера `bck` имеет значение, определенное реализацией по умолчанию. По умолчанию используется `hsh=std::hash<T>`, `keq=std::equal_to<T>` и `alc=std::allocator<T>`
`unordered_set<T>(..., [bck], [hsh], [keq], [alc])`	Выполняет фигурную инициализацию вновь созданного неупорядоченного множества
`unordered_set<T>(beg, end [bck], [hsh], [keq], [alc])`	Создает неупорядоченное множество с элементами в полуоткрытом диапазоне от `beg` до `end`
`unordered_set<T>(s)`	Глубоко копирует `s`; выделяет новую память
`unordered_set<T>(move(s))`	Принимает владение элементами в `s`. Нет выделений памяти

Можно проверить количество сегментов в `unordered_set`, используя метод `bucket_count`. Также можно получить максимальное количество сегментов, используя метод `max_bucket_count`.

Важным понятием производительности `unordered_set` во время выполнения является его *коэффициент загрузки*, среднее количество элементов в сегменте. Можно получить коэффициент загрузки `unordered_set` с помощью метода `load_factor`, который эквивалентен `size()`, деленному на `bucket_count()`. Каждый `unordered_set` имеет максимальный коэффициент загрузки, который вызывает увеличение количества сегментов и потенциально дорогое повторное рехеширование всех содержащихся элементов. *Рехеширование* — это операция, при которой элементы реорганизуются в новые сегменты. Это требует генерации новых хешей для каждого элемента, что может быть относительно затратной вычислительной операцией.

Можно получить максимальный коэффициент загрузки, используя `max_load_factor`, который перегружен, поэтому можно установить новый максимальный коэффициент загрузки (по умолчанию он равен 1,0).

Чтобы избежать затратного рехеширования в неподходящее время, можно провести рехеширование вручную, используя метод `rehash`, который принимает аргумент `size_t` для желаемого количества сегментов. Также можно использовать метод `reserve`, который вместо этого принимает аргумент `size_t` для желаемого количества *элементов*.

В листинге 13.31 приведены некоторые из основных операций управления сегментами.

Создается `unordered_set` и определяется количество сегментов 100 ❶. Это приводит к тому, что значение `bucket_count` составляет не менее 100 ❷, что должно быть меньше или равно значению `max_bucket_count` ❸. По умолчанию `max_load_factor` составляет 1.0 ❹.

Листинг 13.31. Управление сегментами unordered_set

```
#include <unordered_set>
TEST_CASE("std::unordered_set") {
  std::unordered_set<unsigned long> sheep(100); ❶
  SECTION("allows bucket count specification on construction") {
    REQUIRE(sheep.bucket_count() >= 100); ❷
    REQUIRE(sheep.bucket_count() <= sheep.max_bucket_count()); ❸
    REQUIRE(sheep.max_load_factor() == Approx(1.0)); ❹
  }
  SECTION("allows us to reserve space for elements") {
    sheep.reserve(100'000); ❺
    sheep.insert(0);
    REQUIRE(sheep.load_factor() <= 0.00001); ❻

    while(sheep.size() < 100'000)
      sheep.insert(sheep.size()); ❼
    REQUIRE(sheep.load_factor() <= 1.0); ❽
  }
}
```

В следующем тесте вызывается `reserve` с достаточным пространством для ста тысяч элементов ❺. После вставки элемента `load_factor` должен быть меньше или равен одной стотысячной (0,00001) ❻, потому было зарезервировано достаточно места для ста тысяч элементов. Пока этот порог не перейден, рехеширование не понадобится. После вставки ста тысяч элементов ❼ `load_factor` должен быть меньше или равен 1 ❽. Это демонстрирует, что не нужно проводить рехеширование благодаря `reserve`.

Неупорядоченные мультимножества

Доступный в заголовке STL `<unordered_set>` `std::unordered_multiset` является ассоциативным контейнером, который содержит несортированные, *неуникальные* ключи. `unordered_multiset` поддерживает все те же конструкторы и операции, что и `unordered_set`, но будет хранить избыточные элементы. Это отношение аналогично `unordered_sets` и множествам: и `equal_range`, и `count` ведут себя немного по-разному, чтобы учесть неуникальность ключей.

ПРИМЕЧАНИЕ

Boost также предоставляет `boost::unordered_multiset` в заголовке `<boost/unordered_set.hpp>`.

Ассоциативные массивы

`std::map` (ассоциативный массив), доступный в заголовке `<map>` в STL, является ассоциативным контейнером, который содержит пары ключ-значение. Ключи `map` отсортированы и уникальны, и `map` поддерживает все те же операции, что и `set`. Фактически можно представить `set` как особый вид `map`, содержащий ключи и пус-

тые значения. Соответственно `map` поддерживает эффективную вставку, удаление и поиск, и можно управлять сортировкой с помощью объектов сравнения.

Основным преимуществом работы с ассоциативным массивом вместо набора пар является то, что `map` работает как *ассоциативный массив*. Ассоциативный массив принимает ключ, а не целочисленный индекс. Подумайте, как вы используете методы `at` и `operator[]` для доступа к индексам в последовательных контейнерах. Поскольку последовательные контейнеры имеют естественный порядок элементов, используется целое число для ссылки на них. Ассоциативный массив позволяет использовать типы, отличные от целых, для ссылок на элементы. Например, можно использовать строку или `float` в качестве ключа.

Чтобы включить операции с ассоциативным массивом, `map` поддерживает ряд полезных операций, например позволяя вставлять, изменять и извлекать значения по их связанным ключам.

Создание

Шаблон класса `map<Key, Value, Comparator, Allocator>` принимает четыре параметра шаблона. Первый — тип ключа `Key`. Второй — тип значения `Value`. Третий — тип сравнения, который по умолчанию равен `std::less`. Четвертый параметр — это тип распределителя, который по умолчанию равен `std::allocator<T>`.

Конструкторы ассоциативного массива являются прямыми аналогами конструкторов множества: конструктор по умолчанию, который инициализирует пустой `map`; конструкторы переноса и копирования с обычным поведением; конструктор диапазона, который копирует элементы из диапазона в ассоциативный массив; фигурный инициализатор. Основное отличие заключается в фигурном инициализаторе, потому что нужно инициализировать пары ключ-значение, а не просто ключи. Для достижения этой вложенной инициализации используются вложенные списки инициализаторов, как показано в листинге 13.32.

Листинг 13.32. std::map поддерживает конструктор по умолчанию и фигурную инициализацию

```
#include <map>

auto colour_of_magic = "Colour of Magic";
auto the_light_fantastic = "The Light Fantastic";
auto equal_rites = "Equal Rites";
auto mort = "Mort";

TEST_CASE("std::map supports") {
  SECTION("default construction") {
    std::map<const char*, int> emp; ❶
    REQUIRE(emp.empty()); ❷
  }
  SECTION("braced initialization") {
    std::map<const char*, int> pub_year { ❸
      { colour_of_magic, 1983 }, ❹
```

```
      { the_light_fantastic, 1986 },
      { equal_rites, 1987 },
      { mort, 1987 },
    };
    REQUIRE(pub_year.size() == 4); ❺
  }
}
```

Здесь по умолчанию создается `map` с ключами типа `const char*` и значениями типа `int` ❶. Это приводит к созданию пустого `map` ❷. Во втором тесте снова создается `map` с ключами типа `const char*` и значениями типа `int` ❸, но на этот раз используется фигурная инициализация ❹ для упаковки четырех элементов в `map` ❺.

Семантика перемещения и копирования

Семантика перемещения и копирования `map` идентична семантике `set`.

Модель хранения

И `map`, и `set` используют одну и ту же структуру красно-черного дерева.

Доступ к элементам

Основным преимуществом использования `map` вместо `set` объектов `pair` является то, что `map` предлагает две операции с ассоциативным массивом: `operator[]` и `at`. В отличие от последовательных контейнеров, поддерживающих такие операции, как `vector` и `array`, которые принимают аргумент индекса `size_t`, `map` принимает аргумент `Key` и возвращает ссылку на соответствующее значение. Как и в случае с последовательными контейнерами, `at` выдаст исключение `std::out_of_range`, если данный `key` не существует в `map`. В отличие от последовательных контейнеров, `operator[]` не вызовет неопределенного поведения, если ключ не существует; вместо этого он (по умолчанию) создаст `Value` и вставит соответствующую пару ключ-значение в ассоциативный массив, даже если вы только намеревались выполнить чтение, как показано в листинге 13.33.

Листинг 13.33. std::map — это ассоциативный массив с несколькими методами доступа к элементам

```
TEST_CASE("std::map is an associative array with") {
  std::map<const char*, int> pub_year { ❶
    { colour_of_magic, 1983 },
    { the_light_fantastic, 1986 },
  };
  SECTION("operator[]") {
    REQUIRE(pub_year[colour_of_magic] == 1983); ❷

    pub_year[equal_rites] = 1987; ❸
    REQUIRE(pub_year[equal_rites] == 1987); ❹

    REQUIRE(pub_year[mort] == 0); ❺
  }
```

```
  SECTION("an at method") {
    REQUIRE(pub_year.at(colour_of_magic) == 1983); ❻

    REQUIRE_THROWS_AS(pub_year.at(equal_rites), std::out_of_range); ❼
  }
}
```

Создается ассоциативный массив с названием `pub_year`, содержащий два элемента ❶. Затем используется `operator[]` для извлечения значения, соответствующего ключу `colour_of_magic` ❷. `operator[]` также используется для вставки новой пары ключ-значение `equal_rites`, 1987 ❸ и последующего ее извлечения ❹. Обратите внимание, что при попытке получить элемент с ключом `mort` (которого не существует) ассоциативный массив по умолчанию автоматически инициализирует `int` ❺.

Используя `at`, все равно можно установить и получить ❻ элементы, но при попытке получить доступ к несуществующему ключу будет сгенерировано исключение `std::out_of_range` ❼.

`map` поддерживает все операции поиска элементов, подобные операциям в `set`. Например, `map` поддерживает `find`, который принимает аргумент `key` и возвращает итератор, указывающий на пару ключ-значение, или, если не найдено соответствующего ключа, на конец `map`. Также аналогичным образом поддерживаются `count`, `equal_range`, `lower_bound` и `upper_bound`.

Добавление элементов

В дополнение к методам доступа к элементам `operator[]` и `at` также используются все методы `insert` и `emplace`, доступные из `set`. Нужно просто обрабатывать каждую пару ключ-значение как `std::pair<Key, Value>`. Как и в случае с `set`, `insert` возвращает пару, содержащую итератор и `bool`. Итератор указывает на вставленный элемент, а `bool` отвечает, добавил ли `insert` новый элемент (`true`) или нет (`false`), как показано в листинге 13.34.

Листинг 13.34. std::map поддерживает insert для добавления новых элементов

```
TEST_CASE("std::map supports insert") {
  std::map<const char*, int> pub_year; ❶
  pub_year.insert({ colour_of_magic, 1983 }); ❷
  REQUIRE(pub_year.size() == 1); ❸

  std::pair<const char*, int> tlfp{ the_light_fantastic, 1986 }; ❹
  pub_year.insert(tlfp); ❺
  REQUIRE(pub_year.size() == 2); ❻

  auto [itr, is_new] = pub_year.insert({ the_light_fantastic, 9999 }); ❼
  REQUIRE(itr->first == the_light_fantastic);
  REQUIRE(itr->second == 1986); ❽
  REQUIRE_FALSE(is_new); ❾
  REQUIRE(pub_year.size() == 2); ❿
}
```

По умолчанию создается `map` ❶ и используете метод `insert` с фигурным инициализатором для `pair` ❷. Эта конструкция примерно эквивалентна следующей:

```
pub_year.insert(std::pair<const char*, int>{ colour_of_magic, 1983 });
```

После вставки `map` теперь содержит один элемент ❸. Затем создается отдельная пара ❹ и передается в качестве аргумента в `insert` ❺. Это вставит копию в `map`, поэтому теперь он содержит два элемента ❻.

При попытке вызвать `insert` с новым элементом с тем же ключом `the_light_fantastic` ❼ вы получите итератор, указывающий на элемент, который уже был добавлен ❺. Ключ (`first`) и значение (`second`) совпадают ❽. Возвращаемое значение `is_new` указывает, что новый элемент не был вставлен ❾ и все еще есть два элемента ❿. Это поведение отражает поведение `insert` в `set`.

`map` также предлагает метод `insert_or_assign`, который, в отличие от `insert`, будет перезаписывать существующее значение. Также в отличие от `insert`, `insert_or_assign` принимает отдельные аргументы ключ и значение, как показано в листинге 13.35.

Листинг 13.35. std::map поддерживает insert_or_assign для перезаписи существующих элементов.

```
TEST_CASE("std::map supports insert_or_assign") {
  std::map<const char*, int> pub_year{ ❶
    { the_light_fantastic, 9999 }
  };
  auto [itr, is_new] = pub_year.insert_or_assign(the_light_fantastic, 1986); ❷
  REQUIRE(itr->second == 1986); ❸
  REQUIRE_FALSE(is_new); ❹
}
```

Создается `map` с одним элементом ❶, и затем вызывается `insert_or_assign`, чтобы переназначить значение, связанное с ключом `the_light_fantastic`, на 1986 ❷. Итератор указывает на существующий элемент, и при запросе соответствующего значения с помощью `second` вы заметите, что значение обновлено до 1986 г. ❸. Возвращаемое значение `is_new` также указывает, что был обновлен существующий элемент, а не вставлен новый ❹.

Удаление элементов

Как и `set`, `map` поддерживает `erase` и `clear` для удаления элементов, как показано в листинге 13.36.

Листинг 13.36. std::map поддерживает удаление элементов

```
TEST_CASE("We can remove std::map elements using") {
    std::map<const char*, int> pub_year {
      { colour_of_magic, 1983 },
      { mort, 1987 },
    }; ❶
  SECTION("erase") {
    pub_year.erase(mort); ❷
    REQUIRE(pub_year.find(mort) == pub_year.end()); ❸
```

```
    }
    SECTION("clear") {
      pub_year.clear(); ❹
      REQUIRE(pub_year.empty()); ❺
    }
  }
```

Создается `map` с двумя элементами ❶. В первом тесте вызывается `erase` для элемента с ключом `mort` ❷, поэтому при попытке найти его вы получите `end` ❸. Во втором тесте `map` очищается ❹, в результате чего `empty` возвращает `true` ❺.

Список поддерживаемых операций

Таблица 13.12 обобщает поддерживаемые операции `map`. Ключ `k` имеет тип `K`. Значение `v` имеет тип `V`. `P` — тип `pair<K, V>`, а `p` — тип `P`. Ассоциативный массив `m` — это `map<K, V>`. Кинжал (†) обозначает метод, который возвращает `std::pair<Iterator, bool>`, где итератор указывает на результирующий элемент, а значение `bool` равно `true`, если метод вставил элемент, и `false`, если элемент уже существовал.

Таблица 13.12. Неполный список поддерживаемых операций с `map`

Операция	Примечания
`map<T>{..., [cmp], [alc] }`	Выполняет скрытую инициализацию вновь созданного `map`. По умолчанию используется `cmp=std::less<T>` и `alc=std::allocator <T>`
`map<T>{ beg, end, [cmp], [alc] }`	Конструктор диапазона, который копирует элементы из полуоткрытого диапазона от `beg` до `end`. По умолчанию используется `cmp=std::less<T>` и `alc=std::allocator<T>`
`map<T>(m)`	Глубоко копирует `m`; выделяет новую память
`map<T>(move(m))`	Принимает владение памятью для элементов в `m`. Не выделяет память
`~map`	Уничтожает все элементы, содержащиеся в `map`, и освобождает динамическую память
`m1 = m2`	Уничтожает элементы `m1`; копирует каждый элемент `m2`. Выделяет память только в том случае, если необходимо изменить размер для соответствия элементам `m2`
`m1 = move(m2)`	Уничтожает элементы `m1`; перемещает каждый элемент `m2`. Выделяет память только в том случае, если необходимо изменить размер, чтобы соответствовать элементам `m2`
`m.at(k)`	Получает доступ к значению, соответствующему ключу `k`. Выдает `std::out_of_range`, если ключ не найден
`m[k]`	Получает доступ к значению, соответствующему ключу `k`. Если ключ не найден, вставляет новую пару ключ-значение, используя `k` и инициализированное значение по умолчанию
`m.begin()`	Возвращает итератор, указывающий на первый элемент

Операция	Примечания
`m.cbegin()`	Возвращает итератор `const`, указывающий на первый элемент
`m.end()`	Возвращает итератор, указывающий на элемент, следующий за последним элементом
`m.cend()`	Возвращает итератор `const`, указывающий на элемент, следующий за последним элементом
`m.find(k)`	Возвращает итератор, указывающий на элемент, соответствующий `k`, или `m.end()`, если такого элемента не существует
`m.count(k)`	Возвращает 1, если `map` содержит `k`; в противном случае 0
`m.equal_range(k)`	Возвращает пару итераторов, соответствующих полуоткрытому диапазону элементов, соответствующих `k`
`m.lower_bound(k)`	Возвращает итератор, указывающий на первый элемент не меньший, чем `k`, или `t.end()`, если такого элемента не существует
`m.upper_bound(k)`	Возвращает итератор, указывающий на первый элемент больший, чем `k`, или `t.end()`, если такого элемента не существует
`m.clear()`	Удаляет все элементы из `map`
`m.erase(k)`	Удаляет элемент с ключом `k`
`m.erase(itr)`	Удаляет элемент, на который указывает `itr`
`m.erase(beg, end)`	Удаляет все элементы в полуоткрытом диапазоне от `beg` до `end`
`m.insert(p)`	Вставляет копию пары `p` в `map` †
`m.insert_or_assign(k, v)`	Если `k` существует, заменяет соответствующее значение на `v`. Если `k` не существует, вставляет пару `k`, `v` на карту †
`m.emplace(...)`	Создает `P` на месте путем передачи аргументов... †
`m.emplace_ hint(itr,...)`	Создает `P` на месте путем передачи аргументов.... Использует `itr` в качестве подсказки, куда вставить новый элемент †
`m.try_emplace(k,...)`	Если ключ `k` существует, ничего не делает. Если `k` не существует, создает `V` на месте путем передачи аргументов...
`m.empty()`	Возвращает `true`, если размер `map` равен нулю; иначе — `false`
`m.size()`	Возвращает количество элементов в `map`
`m.max_size()`	Возвращает максимальное количество элементов в `map`
`m.extract(k)` `m.extract(itr)`	Получает дескриптор узла, которому принадлежит элемент, соответствующий `k` или на который указывает `itr`. (Это единственный способ удалить элемент только для перемещения)
`m1.merge(m2)` `m1.merge(move(m2))`	Объединяет каждый элемент `m2` с `m1`. Если аргумент является `r`-значением, перемещает элементы в `m1`
`m1.swap(m2)` `swap(m1, m2)`	Заменяет все элементы `m1` на элементы `m2`

Мультиассоциативные массивы

`std::multimap`, доступный в заголовке `<map>` в STL, является ассоциативным контейнером, который содержит пары ключ-значение с *неуникальными* ключами. Поскольку ключи не являются уникальными, `multimap` не поддерживает функции ассоциативного массива, которые поддерживает `map`. А именно, `operator[]` и `at` не поддерживаются. Как и в случае с мультимножеством, `multimap` предлагает доступ к элементам в основном через метод `equal_range`, как показано в листинге 13.37.

Листинг 13.37. std::multimap поддерживает неуникальные ключи

```
TEST_CASE("std::multimap supports non-unique keys") {
  std::array<char, 64> far_out {
    "Far out in the uncharted backwaters of the unfashionable end..."
  }; ❶
  std::multimap<char, size_t> indices; ❷
  for(size_t index{}; index<far_out.size(); index++)
    indices.emplace(far_out[index], index); ❸

  REQUIRE(indices.count('a') == 6); ❹

  auto [itr, end] = indices.equal_range('d'); ❺
  REQUIRE(itr->second == 23); ❻
  itr++;
  REQUIRE(itr->second == 59); ❼
  itr++;
  REQUIRE(itr == end);
}
```

Создается `array`, содержащий сообщение ❶. Также по умолчанию создается `multimap <char, size_t>` под названием `indices`, который будет использоваться для хранения индекса каждого символа в сообщении ❷. Перебирая массив, можно сохранить каждый символ в сообщении вместе с его индексом в виде нового элемента в `multimap` ❸. Поскольку разрешено иметь неуникальные ключи, можно использовать метод `count`, чтобы узнать, сколько индексов добавлено с помощью ключа `a` ❹. Также можно использовать метод `equal_range` для получения полуоткрытого диапазона индексов с ключом `d` ❺. Используя результирующие итераторы `begin` и `end`, можно увидеть, что сообщение имеет букву `d` с индексами 23 ❻ и 59 ❼.

Помимо `operator[]` и `at`, каждая операция из таблицы 13.12 также работает для `multimap`. (Обратите внимание, что метод `count` может принимать значения, отличные от 0 и 1.)

Неупорядоченные ассоциативные и мультиассоциативные массивы

Неупорядоченные ассоциативные и мультиассоциативные массивы полностью аналогичны неупорядоченным множествам и мультимножествам. `std::unordered_map` и `std::unordered_multimap` и доступны в заголовке `<unordered_map>` в STL. Эти ассоциативные контейнеры обычно используют красно-черное дерево, как и их установленные аналоги. Им также требуются хеш-функция и функция эквивалентности, и они поддерживают интерфейс сегмента.

ПРИМЕЧАНИЕ

Boost предлагает `boost::unordered_map` и `boost::unordered_multimap` в заголовке `<boost/unordered_map.hpp>`.

Специализированные ассоциативные контейнеры

Используйте `set`, `map` и связанные с ними неуникальные и неупорядоченные аналоги в качестве вариантов по умолчанию, когда нужны ассоциативные структуры данных. При возникновении особых потребностей библиотеки Boost предлагают ряд специализированных ассоциативных контейнеров, как показано в таблице 13.13.

Таблица 13.13. Специальные контейнеры Boost

Класс/Заголовок	Описание
`boost::container::flat_map` `<boost/container/flat_map.hpp>`	Похож на `map` в STL, но реализован как упорядоченный вектор. Это означает наличие быстрого доступа к случайному элементу
`boost::container::flat_set` `<boost/container/flat_set.hpp>`	Аналогично `set` в STL, но он реализован как упорядоченный вектор. Это означает наличие быстрого доступа к случайному элементу
`boost::intrusive::*` `<boost/intrusive/*.hpp>`	Навязчивые контейнеры предъявляют требования к элементам, которые они содержат (например, наследование от определенного базового класса). Взамен они предлагают существенное повышение производительности
`boost::multi_index_container` `<boost/multi_index_container.hpp>`	Позволяет создавать ассоциативные массивы, используя несколько индексов, а не один (подобно `map`)
`boost::ptr_map` `boost::ptr_set` `boost::ptr_unordered_map` `boost::ptr_unordered_set` `<boost/ptr_container/*.hpp>`	Наличие коллекции умных указателей может быть неоптимальным. Векторы-указатели управляют коллекцией динамических объектов более эффективным и удобным способом
`boost::bimap` `<boost/bimap.hpp>`	`bimap` — это ассоциативный контейнер, который позволяет использовать оба типа в качестве ключа
`boost::heap::binomial_heap` `boost::heap::d_ary_heap` `boost::heap::fibonacci_heap` `boost::heap::pairing_heap` `boost::heap::priority_queue` `boost::heap::skew_heap` `<boost/heap/*.hpp>`	Контейнеры Boost `Heap` реализуют более продвинутые многофункциональные версии `priority_queue`

Графы и деревья свойств

В этом разделе рассматриваются две специализированные библиотеки Boost, которые служат специфическим, но важным целям: моделированию графов и деревьев свойств. *Граф* — это набор объектов, где некоторые из них имеют попарное отношение. Объекты называются *вершинами*, а их отношения называются *ребрами*. На рисунке 13.3 показан граф, содержащий четыре вершины и пять ребер.

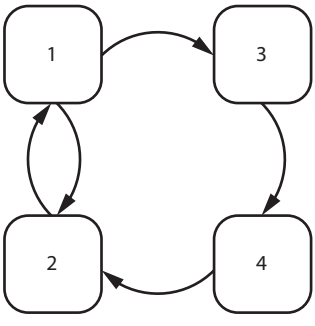


Рис. 13.3. Граф, содержащий четыре вершины и пять ребер

Каждый квадрат представляет вершину, а каждая стрелка — ребро.

Дерево свойств — это древовидная структура, хранящая вложенные пары ключ—значение. Иерархическая природа пар ключ—значение дерева свойств делает его гибридом между картой и графом; каждая пара ключ—значение имеет отношение к другим парам ключ—значение. На рис. 13.4 приведен пример дерева свойств, содержащего вложенные пары ключ—значение.

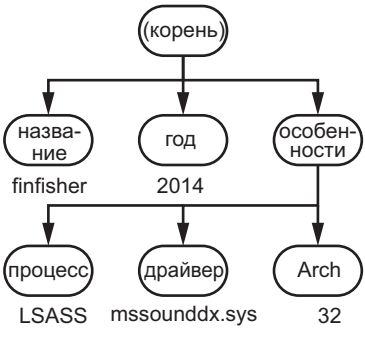


Рис. 13.4. Пример дерева свойств

Корневой элемент имеет три дочерних элемента: название, год и особенности. На рис. 13.4 используется название `finfisher`, год — `2014`, а особенности имеют три дочерних элемента: процесс со значением LSASS, драйвер со значением `mssounddx.sys` и `Arch` со значением 32.

Библиотека Boost Graph

Boost Graph Library (BGL) — это набор коллекций и алгоритмов для хранения и управления графами. BGL предлагает три контейнера, которые представляют графы:

- `boost::adjacency_list` в заголовке `<boost/graph/adjacency_list.hpp>`;
- `boost::adjacency_matrix` в заголовке `<boost/graph/adjacency_matrix.hpp>`;
- `boost::edge_list` в заголовке `<boost/graph/edge_list.hpp>`.

Для построения графов используются две функции, не являющиеся членами: `boost::add_vertex` и `boost::add_edge`. Чтобы добавить вершину в один из контейнеров графа BGL, объект графа передается в `add_vertex`, который будет возвращать ссылку на новый объект вершины. Чтобы добавить ребро, в `add_edge` передаются исходная вершина, конечная вершина, а затем граф.

BGL содержит ряд алгоритмов, специфических для графов. Можно подсчитать количество вершин в объекте графа, передав его функции `boost::num_vertices` и количеству ребер с помощью `boost::num_edges`. Также можно запросить граф смежных вершин. Две вершины являются непрерывными, если они имеют общее ребро. Чтобы получить вершины, смежные с конкретной вершиной, можно передать ее и объект графа функции, не являющейся членом, `boost::acent_vertices`. Это возвращает полуоткрытый диапазон как `std::pair` итераторов.

В листинге 13.38 показано, как можно построить граф из рис. 13.3, подсчитать его вершины и ребра и вычислить смежные вершины.

Листинг 13.38. boost::adjacency_list хранит данные графа

```
#include <set>
#include <boost/graph/adjacency_list.hpp>

TEST_CASE("boost::adjacency_list stores graph data") {
  boost::adjacency_list<> graph{}; ❶
  auto vertex_1 = boost::add_vertex(graph);
  auto vertex_2 = boost::add_vertex(graph);
  auto vertex_3 = boost::add_vertex(graph);
  auto vertex_4 = boost::add_vertex(graph); ❷
  auto edge_12 = boost::add_edge(vertex_1, vertex_2, graph);
  auto edge_13 = boost::add_edge(vertex_1, vertex_3, graph);
  auto edge_21 = boost::add_edge(vertex_2, vertex_1, graph);
  auto edge_24 = boost::add_edge(vertex_2, vertex_4, graph);
  auto edge_43 = boost::add_edge(vertex_4, vertex_3, graph); ❸

  REQUIRE(boost::num_vertices(graph) == 4); ❹
  REQUIRE(boost::num_edges(graph) == 5); ❺

  auto [begin, end] = boost::adjacent_vertices(vertex_1, graph); ❻
  std::set<decltype(vertex_1)> neighboors_1 { begin, end }; ❼
  REQUIRE(neighboors_1.count(vertex_2) == 1); ❽
  REQUIRE(neighboors_1.count(vertex_3) == 1); ❾
  REQUIRE(neighboors_1.count(vertex_4) == 0); ❿
}
```

Здесь создается `adjacency_list` под названием `graph` ❶, а затем добавляются четыре вершины, используя `add_vertex` ❷. Затем были добавлены все ребра, представленные на рисунке 13.3 с помощью `add_edge` ❸. `num_vertices` показывает, что были добавлены четыре вершины ❹, а `num_edges` сообщает, что были добавлены пять ребер ❺.

Наконец, были определены смежные вершины для `vertex_1`, которые распаковываются в итераторы `begin` и `end` ❻. Эти итераторы используются для создания `std::set` ❼, который применяется, чтобы показать, что вершины `vertex_2` ❽ и `vertex_3` ❾ являются непрерывными, но `vertex_4` — не смежная ❿.

Деревья свойств в Boost

Boost предлагает `boost::property_tree::ptree` в заголовке <`boost/property_tree/ptree.hpp`>. Это дерево свойств, которое позволяет создавать и запрашивать деревья свойств, а также применять ограниченную сериализацию для различных форматов.

Дерево `ptree` является конструируемым по умолчанию. Конструктор по умолчанию создаст пустое дерево.

Можно вставить элементы в `ptree`, используя метод `put`, который принимает путь и аргумент значения. *Путь* — это последовательность из одного или нескольких вложенных ключей, разделенных точкой (.), и *значение* — это объект произвольно типизированного типа.

Можно получить поддерево из `ptree` с помощью метода `get_child`, который принимает путь к желаемому поддереву. Если у поддерева нет дочерних элементов (так называемый *листовой узел*), также можно использовать шаблон метода `get_value` для извлечения соответствующего значения из пары ключ-значение; `get_value` принимает один параметр шаблона, соответствующий желаемому типу вывода.

Наконец, `ptree` поддерживает сериализацию и десериализацию в несколько форматов, включая нотацию объектов JavaScript (JSON), формат файла инициализации Windows (INI), расширяемый язык разметки (XML) и пользовательский формат, специфичный для `ptree`, называемый INFO. Например, чтобы записать `ptree` в файл в формате JSON, можно использовать функцию `boost::property_tree::write_json` из заголовка <`boost/property_tree/json_parser.hpp`>. Функция `write_json` принимает два аргумента: путь к желаемому выходному файлу и ссылку на `ptree`.

В листинге 13.39 показаны эти основные функции `ptree` путем создания `ptree`, представляющего дерево свойств на рисунке 13.4, с записью `ptree` в файл как JSON, и его последующего чтения.

Листинг 13.39. Метод boost::property_tree::ptree хранит данные дерева. Вывод показывает содержимое rootkit.json.

```
#include <boost/property_tree/ptree.hpp>
#include <boost/property_tree/json_parser.hpp>
```

```
TEST_CASE("boost::property_tree::ptree stores tree data") {
  using namespace boost::property_tree;
  ptree p; ❶
  p.put("name", "finfisher");
  p.put("year", 2014);
  p.put("features.process", "LSASS");
  p.put("features.driver", "mssounddx.sys");
  p.put("features.arch", 32); ❷
  REQUIRE(p.get_child("year").get_value<int>() == 2014); ❸

  const auto file_name = "rootkit.json";
  write_json(file_name, p); ❹

  ptree p_copy;
  read_json(file_name, p_copy); ❺
  REQUIRE(p_copy == p); ❻
}
-------------------------------------------------------------------------------
{
    "name": "finfisher",
    "year": "2014",
    "features": {
        "process": "LSASS",
        "driver": "mssounddx.sys",
        "arch": "32"
    }
} ❹
```

Здесь по умолчанию создается метод `ptree` ❶, который заполняется значениями ключей, показанными на рисунке 13.4. Ключи с родителями, например `arch` ❷, используют точки, чтобы показать соответствующий путь. Используя `get_child`, извлекается поддерево для ключа `year`. Поскольку это листовой узел (не имеющий дочерних элементов), вызывается `get_value`, тип вывода указывается как `int` ❸.

Затем JSON-представление `ptree` записывается в файл `rootkit.json` ❹. Чтобы убедиться, что было возвращено то же дерево свойств, по умолчанию создается другое `ptree` с именем `p_copy` и передается в `read_json` ❺. Эта копия эквивалентна оригиналу ❻, показывающему, что операция сериализаци/десериализации прошла успешно.

Списки инициализаторов

Можно принять списки инициализаторов в пользовательских типах, включив контейнер `std::initializer_list`, доступный в заголовке `<initializer_list>` в STL. `initializer_list` — это шаблон класса, который принимает один параметр шаблона, соответствующий базовому типу, содержащемуся в списке инициализатора. Этот шаблон служит простым прокси для доступа к элементам списка инициализатора.

`initializer_list` является неизменным и поддерживает следующие операции:

- метод `size` возвращает количество элементов в `initializer_list`;
- методы `begin` и `end` возвращают обычные итераторы с полуоткрытым диапазоном.

Как правило, нужно разрабатывать функции для принятия `initializer_list` по значению.

В листинге 13.40 реализован класс `SquareMatrix`, в котором хранится матрица с равным количеством строк и столбцов. Внутри класс будет содержать элементы в векторе векторов.

Листинг 13.40. Реализация SquareMatrix

```
#include <cmath>
#include <stdexcept>
#include <initializer_list>
#include <vector>

size_t square_root(size_t x) { ❶
  const auto result = static_cast<size_t>(sqrt(x));
  if (result * result != x) throw std::logic_error{ "Not a perfect square." };
  return result;
}

template <typename T>
struct SquareMatrix {
  SquareMatrix(std::initializer_list<T> val) ❷
    : dim{ square_root(val.size()) }, ❸
      data(dim, std::vector<T>{}) { ❹
    auto itr = val.begin(); ❺
    for(size_t row{}; row<dim; row++){
      data[row].assign(itr, itr+dim); ❻
      itr += dim; ❼
    }
  }
  T& at(size_t row, size_t col) {
    if (row >= dim || col >= dim)
      throw std::out_of_range{ "Index invalid." }; ❽
    return data[row][col]; ❾
  }
  const size_t dim;
private:
  std::vector<std::vector<T>> data;
};
```

Здесь объявляется удобная функция `square_root`, которая находит квадратный корень из `size_t`, и выдает исключение, если аргумент не является идеальным квадратом ❶. Шаблон класса `SquareMatrix` определяет один конструктор, который принимает `std::initializer` с именем `val` ❷. Это разрешает инициализацию в фигурном выражении.

Во-первых, нужно определить размеры `SquareMatrix`. Используйте функцию `square_root`, чтобы вычислить квадратный корень из `val.size()` ❸ и сохранить его в поле

dim, которое представляет количество строк и столбцов экземпляра SquareMatrix. Затем можно использовать dim для инициализации вектора векторов, используя его конструктор заполнения ❹. Каждый из этих векторов будет соответствовать строке в SquareMatrix. Затем извлекается итератор, указывающий на первый элемент в initializer_list ❺. Каждая строка в SquareMatrix перебирается, назначая соответствующий вектор соответствующему полуоткрытому диапазону ❻. Итератор увеличивается на каждой итерации, чтобы указывать на следующую строку ❼.

Наконец, реализуется метод at для разрешения доступа к элементу. Выполняется проверка границ ❽, а затем возвращается ссылка на нужный элемент путем извлечения соответствующего vector и элемент ❾.

В листинге 13.41 показано, как использовать фигурную инициализацию для генерации объекта SquareMatrix.

Листинг 13.41. Использование фигурных инициализаторов с SquareMatrix

```
TEST_CASE("SquareMatrix and std::initializer_list") {
  SquareMatrix<int> mat { ❶
     1, 2, 3, 4,
     5, 0, 7, 8,
     9, 10, 11, 12,
    13, 14, 15, 16
  };
  REQUIRE(mat.dim == 4); ❷
  mat.at(1, 1) = 6; ❸
  REQUIRE(mat.at(1, 1) == 6); ❹
  REQUIRE(mat.at(0, 2) == 3); ❺
}
```

Фигурные инициализаторы используются для настройки SquareMatrix ❶. Поскольку список инициализаторов содержит 16 элементов, получается dim со значением 4 ❷. Можно использовать at для получения ссылки на любой элемент, то есть можно установить ❸ и получить элементы ❹❺.

Итоги

Эта глава началась с обсуждения двух контейнеров последовательностей, массива и вектора, которые предлагают отличный баланс между производительностью и функциональностью в широком спектре приложений. Затем вы узнали о нескольких контейнерах последовательностей: deque, list, stack, queue, priority_queue и bitset, которые заполняются, когда vector не удовлетворяет требованиям конкретного приложения. Затем вы изучили основные ассоциативные контейнеры, set и map, а также их неупорядоченные и мультиварианты. Вы также узнали о двух специализированных контейнерах Boost, graph и ptree. Глава завершилась кратким обсуждением включения initializer_lists в пользовательские типы.

Упражнения

13.1. Напишите программу, которая по умолчанию создает `std::vector` с типами `unsigned long`. Выведите емкость вектора, а затем зарезервируйте 10 элементов. Затем добавьте первые 20 элементов ряда Фибоначчи к вектору. Выведите емкость снова. Соответствует ли емкость количеству элементов в векторе? Почему да или почему нет? Выведите элементы `vector`, используя основанный на диапазоне цикл `for`.

13.2. Перепишите листинги 2.9, 2.10 и 2.11 в главе 2, используя `std::array`.

13.3. Напишите программу, которая принимает любое количество аргументов командной строки и печатает их в алфавитно-цифровом порядке. Используйте `std::set<const char*>` для хранения элементов, затем выполните перебор `set` для получения отсортированного результата. Нужно реализовать собственный оператор сравнения, который сравнивает две строки в стиле C.

13.4. Рассмотрим следующую программу, которая описывает производительность функции, суммирующей ряд Фибоначчи:

```
#include <chrono>
#include <cstdio>
#include <random>

long fib_sum(size_t n) {❶
  // TODO: Адаптировать код из упражнения 13.1
  return 0;
}

long random() { ❷
  static std::mt19937_64 mt_engine{ 102787 };
  static std::uniform_int_distribution<long> int_d{ 1000, 2000 };
  return int_d(mt_engine);
}

struct Stopwatch { ❸
  Stopwatch(std::chrono::nanoseconds& result)
    : result{ result },
    start{ std::chrono::system_clock::now() } { }
  ~Stopwatch() {
    result = std::chrono::system_clock::now() - start;
  }
private:
  std::chrono::nanoseconds& result;
  const std::chrono::time_point<std::chrono::system_clock> start;
};

long cached_fib_sum(size_t n) { ❹
```

```
  static std::map<long, long> cache;
  // // TODO: Реализовать функцию
  return 0;
}

int main() {

  size_t samples{ 1'000'000 };
  std::chrono::nanoseconds elapsed;
  {
   Stopwatch stopwatch{elapsed};
   volatile double answer;
   while(samples--) {
     answer = fib_sum(random());❺
     //answer = cached_fib_sum(random()); ❻
   }
  }
  printf("Elapsed: %g s.\n", elapsed.count() / 1'000'000'000.); ❼
}
```

Эта программа содержит вычислительно интенсивную функцию `fib_sum` ❶, которая вычисляет сумму ряда Фибоначчи с заданной длиной. Адаптируйте код из упражнения 13.1 путем (а) генерации подходящего вектора и (б) суммирования по результату с помощью цикла `for` на основе диапазона. Функция `random` ❷ возвращает случайное число от 1000 до 2000, а класс `Stopwatch` ❸, принятый из листинга 12.25 в главе 12, помогает определить истекшее время. В `main` программы выполняется миллион вычислений функции `fib_sum` с использованием случайного ввода ❺. Рассчитайте, сколько времени это займет, и выведите результат перед выходом из программы ❼. Скомпилируйте программу и запустите ее несколько раз, чтобы понять, сколько времени занимает запуск программы. (Это называется базовой линией.)

13.5. Далее закомментируйте ❺ и раскомментируйте ❻. Реализуйте функцию `cached_fib_sum` ❹, чтобы проверить, была ли вычислена `fib_sum` для заданной длины. (Рассматривайте длину `n` как ключ в кэше.) Если ключ присутствует в кэше, просто верните результат. Если ключ отсутствует, вычислите правильный ответ с помощью `fib_sum`, сохраните новую запись значения ключа в кэш и верните результат. Запустите программу еще раз. Она стала быстрее? Попробуйте `unordered_map` вместо `map`. Можно ли было использовать вместо этого `vector`? Как быстро можно запустить программу?

13.6. Реализуйте класс `Matrix`, например `SquareMatrix`, в листинге 13.38. Ваша матрица должна допускать неравное количество строк и столбцов. Примите в качестве первого аргумента конструктора количество строк в матрице.

Что еще почитать?

- Международный стандарт ИСО/МЭК (2017) — Язык программирования C++ (Международная организация по стандартизации; Женева, Швейцария; isocpp.org/std/the-standard/)
- «The Boost C++ Libraries», 2nd Edition, Boris Schäling (XML Press, 2014)
- «Стандартная библиотека C++. Справочное руководство», 2-е издание, Николай М. Джосаттис (Вильямс, 2017)

14 Итераторы

Скажите «друг» и войдите.

Дж. Р. Р. Толкин, «Властелин колец»

Итераторы являются компонентом STL, который обеспечивает интерфейс между контейнерами и алгоритмами для управления ими. Итератор — это интерфейс к типу, который знает, как пройти конкретную последовательность, и предоставляет простые, похожие на указатель операции для элементов.

Каждый итератор поддерживает как минимум следующие операции:

- доступ к текущему элементу (`operator*`) для чтения и/или записи;
- переход к следующему элементу (`operator++`);
- конструктор копирования.

Итераторы делятся на категории в зависимости от того, какие дополнительные операции они поддерживают. Эти категории определяют, какие алгоритмы доступны и что можно сделать с помощью итератора в общем коде. В этой главе вы узнаете об этих категориях итераторов, вспомогательных функциях и адаптерах.

Категории итераторов

Категория итератора определяет операции, которые он поддерживает. Эти операции включают в себя чтение и запись элементов, перебор в прямом и обратном порядке, многократное чтение и доступ к случайным элементам.

Поскольку код, принимающий итератор, обычно является общим, тип итератора обычно является параметром шаблона, который можно кодировать с помощью

концепций, о которых вы узнали в разделе «Концепты» на с. 226. Хотя, вероятно, не придется взаимодействовать с итераторами напрямую (если вы не пишете библиотеку), все равно нужно знать категории итераторов, чтобы не пытаться применять алгоритм к неподходящим итераторам. Если вы это все-таки сделаете, то, скорее всего, получите загадочные ошибки компилятора. Из раздела «Проверка типов в шаблонах» на с. 224 вы знаете, что из-за способа создания экземпляров шаблонов, сообщения об ошибках, генерируемые из неподходящих аргументов типа, обычно непонятны.

Итераторы вывода

Итератор вывода можно использовать только для записи и увеличения. Представьте итератор вывода как бездонную яму, в которую вы бросаете данные.

При использовании итератора вывода вы пишете, затем увеличиваете, затем записываете, затем увеличиваете, и так до тошноты. После записи в итератор вывода нельзя записывать снова до хотя бы однократного увеличения. Аналогично, после того как вы увеличили итератор вывода, нельзя снова увеличить его перед записью.

Чтобы записать в выходной итератор, разыменуйте итератор с помощью оператора разыменования (*) и присвойте значение результирующей ссылке. Чтобы увеличить выходной итератор, используйте `operator++` или `operator++(int)`.

Если вы не пишете библиотеку C++, то *реализовывать* собственные типы выходных итераторов скорее всего не придется; но *использовать* типы выходных итераторов вы будете часто.

Одним из важных применений является запись в контейнеры, как если бы они были итераторами вывода. Для этого используются итераторы вставки.

Итераторы вставки

Итератор вставки (или *вставщик*) — это итератор вывода, который упаковывает контейнер и преобразует записи (назначения) в вставки. В заголовке `<iterator>` в STL существуют три итератора вставки в качестве шаблонов классов:

- `std::back_insert_iterator`;
- `std::front_insert_iterator`;
- `std::insert_iterator`.

STL также предлагает три вспомогательные функции для создания этих итераторов:

- `std::back_inserter`;
- `std::front_inserter`;
- `std::insertter`.

`back_insert_iterator` преобразует записи итератора в вызовы `push_back` контейнера, тогда как `front_insert_iterator` вызывает `push_front`. Эти итераторы вставки

предоставляют один конструктор, который принимает ссылку на контейнер, а их соответствующие вспомогательные функции принимают один аргумент. Очевидно, что упакованный контейнер должен реализовывать соответствующий метод. Например, вектор не будет работать с `front_insert_iterator`, а `set` не будет работать ни с одним из них.

`insert_iterator` принимает два аргумента конструктора: контейнер для переноса и итератор, указывающий на позицию в этом контейнере. Затем `insert_iterator` преобразует записи в вызовы метода `insert` контейнера и передаст позицию, указанную в конструкторе, в качестве первого аргумента. Например, `insert_iterator` используется для вставки в середину последовательного контейнера или для добавления элементов в `set` с подсказкой.

ПРИМЕЧАНИЕ

Внутренне все итераторы вставки полностью игнорируют `operator++`, `operator++(int)` и `operator*`. Контейнерам не нужен этот промежуточный шаг между вставками, и такое требование типично для выходных итераторов.

В листинге 14.1 показаны основные способы использования трех итераторов вставки путем добавления элементов в `deque`.

Листинг 14.1. Итераторы вставки преобразуют записи во вставки контейнера

```
#include <deque>
#include <iterator>

TEST_CASE("Insert iterators convert writes into container insertions.") {
  std::deque<int> dq;
  auto back_instr = std::back_inserter(dq); ❶
  *back_instr = 2; ❷ // 2
  ++back_instr; ❸
  *back_instr = 4; ❹ // 2 4
  ++back_instr;

  auto front_instr = std::front_inserter(dq); ❺
  *front_instr = 1; ❻ // 1 2 4
  ++front_instr;

  auto instr = std::inserter(dq, dq.begin()+2); ❼
  *instr = 3; ❽ // 1 2 3 4
  instr++;

  REQUIRE(dq[0] == 1);
  REQUIRE(dq[1] == 2);
  REQUIRE(dq[2] == 3);
  REQUIRE(dq[3] == 4); ❾
}
```

Сначала создается `back_insert_iterator` с `back_inserter`, чтобы обернуть `deque` с именем `dq` ❶. При записи в `back_insert_iterator` он преобразует запись в `push_back`,

поэтому в deque содержится один элемент, 2 ❷. Поскольку итераторы вывода требуют увеличения, прежде чем можно будет снова записывать, тут же обрабатывается увеличение ❸. При записи 4 в back_insert_iterator он снова переводит запись в push_back, поэтому в deque содержатся элементы 2 4 ❹.

Затем создается front_insert_iterator с front_inserter для обертки dq ❺. Запись 1 в этот недавно созданный вставщик модуль приводит к вызову push_front, поэтому в deque содержатся элементы 1 2 4 ❻.

Наконец, создается insert_iterator со вставкой, передавая dq и итератор, указывающий на его третий элемент (4). При записи 3 в этот вставщик ❽ он вставляется непосредственно перед элементом, на который указывает итератор, переданный в конструктор ❼. Это приводит к dq, содержащему элементы 1 2 3 4 ❾.

Таблица 14.1 обобщает итераторы вставки.

Таблица 14.1. Список итераторов вставки

Класс	Удобная функция	Делегированная функция	Примеры контейнеров
back_insert_iterator	back_inserter	push_back	vector, deque, list
front_insert_iterator	front_inserter	push_front	deque, list
insert_iterator	inserter	insert	vector, deque, list, set

Список поддерживаемых операций итератора вывода

Таблица 14.2 обобщает поддерживаемые операции итератора вывода.

Таблица 14.2. Поддерживаемые операции итератора вывода

Операция	Примечания
*itr=t	Записывает в выходной итератор. После операции итератор может увеличиваться, но не обязательно разыменовываться
++itr itr++	Увеличивает итератор. После операции итератор может быть разыменован или исчерпан (после достижения конца), но его необязательно увеличивать
iterator-type{ itr }	Создает итератор из itr через конструктор копирования

Итераторы ввода

Можно использовать *итератор ввода* для чтения, увеличения и проверки на равенство. Он является соединением с выходным итератором. Можно перебирать итератор ввода только один раз.

Обычный шаблон при чтении из итератора ввода — получение полуоткрытого диапазона с итераторами *begin* и *end*. Чтобы прочитать диапазон, читаем итератор `begin`, используя `operator*` с последующим увеличением с помощью `operator++`. Затем проверяем, равен ли итератор `end`. Если это так, диапазон исчерпан. Если нет, можно продолжить чтение/увеличение.

ПРИМЕЧАНИЕ

Итераторы ввода — это магия, которая заставляет работать выражения диапазона, о которых мы говорили в разделе «Циклы `for` на основе диапазонов» на с. 302.

Каноническое использование итератора ввода заключается в обертке стандартного ввода программы (обычно с клавиатуры). Как только значение из стандартного ввода будет прочитано, оно исчезнет. Нельзя вернуться к началу и переиграть. Такое поведение очень хорошо соответствует операциям, поддерживаемым итератором ввода.

В «Кратком курсе по итераторам» на с. 491 вы узнали, что каждый контейнер предоставляет итераторы с методами `begin/cbegin/end/cend`. Все эти методы являются *как минимум* итераторами ввода (и они могут поддерживать дополнительную функциональность). Например, в листинге 14.2 показано, как извлечь диапазон из списка `forward_list` и манипулировать итераторами для чтения вручную.

Листинг 14.2. Взаимодействие с итераторами ввода из forward_list

```
#include <forward_list>

TEST_CASE("std::forward_list begin and end provide input iterators") {
  const std::forward_list<int> easy_as{ 1, 2, 3 }; ❶
  auto itr = easy_as.begin(); ❷
  REQUIRE(*itr == 1); ❸
  itr++; ❹
  REQUIRE(*itr == 2);
  itr++;
  REQUIRE(*itr == 3);
  itr++;
  REQUIRE(itr == easy_as.end()); ❺
}
```

В примере создается `forward_list`, содержащий три элемента ❶. Константность контейнера означает, что элементы являются неизменяемыми, поэтому итераторы поддерживают только операции чтения. Итератор извлекается с помощью метода `begin forward_list` ❷. С использованием `operator*` извлекается элемент, на который указывает `itr` ❸, и выполняется обязательное увеличение ❹. Как только диапазон исчерпан чтением/приращением, `itr` равняется `end` в `forward_list` ❺.

Таблица 14.3 обобщает поддерживаемые операции итератора ввода.

Таблица 14.3. Поддерживаемые операции итератора ввода

Операция	Примечания
`*itr`	Разыменовывает член, на который направлен указатель. Может быть только для чтения (необязательно)
`itr->mbr`	Разыменовывает член `mbr` объекта, на который указывает `itr`
`++itr` `itr++`	Увеличивает итератор. После операции итератор может быть разыменован или исчерпан (после достижения конца), но его необязательно увеличивать
`itr1 == itr2` `itr1 != itr2`	Сравнивает, равны ли итераторы (указывающие на один и тот же элемент)
`iterator-type{ itr }`	Создает итератор из `itr` через конструктор копирования

Однонаправленные итераторы

Однонаправленный итератор — это итератор ввода с дополнительными функциями: однонаправленный итератор также может быть пройден несколько раз, также к нему можно применять конструктор по умолчанию и присваивание копии. Можно использовать однонаправленный итератор вместо итератора ввода во всех случаях.

Все контейнеры STL предоставляют однонаправленные итераторы. Соответственно, `forward_list`, используемый в листинге 14.2, фактически предоставляет однонаправленный итератор (который также является итератором ввода).

В листинге 14.3 показан обновленный листинг 14.2: `forward_list` перебирается несколько раз.

Листинг 14.3. Обход однонаправленного итератора дважды

```
TEST_CASE("std::forward_list's begin and end provide forward iterators") {
  const std::forward_list<int> easy_as{ 1, 2, 3 }; ❶
  auto itr1 = easy_as.begin(); ❷
  auto itr2{ itr1 }; ❸
  int double_sum{};
  while (itr1 != easy_as.end()) ❹
    double_sum += *(itr1++);
  while (itr2 != easy_as.end()) ❺
    double_sum += *(itr2++);
  REQUIRE(double_sum == 12); ❻
}
```

Снова создается `forward_list`, содержащий три элемента ❶. Извлекается итератор с именем `itr1` с помощью метода `begin` для `forward_list` ❷, а затем создается копия с именем `itr2` ❸. `itr1` ❹ и `itr2` ❺ исчерпываются дважды, перебирая диапазон и суммируя значения оба раза. Результирующая двойная сумма равна 12 ❻.

Таблица 14.4 обобщает поддерживаемые операции однонаправленного итератора.

Таблица 14.4. Поддерживаемые операции однонаправленного итератора

Операция	Примечания
`*itr`	Разыменовывает член, на который направлен указатель. Может быть только для чтения (необязательно)
`itr->mbr`	Разыменовывает член `mbr` объекта, на который указывает `itr`
`++itr` `itr++`	Увеличивает итератор так, что тот указывает на следующий элемент
`itr1 == itr2` `itr1 != itr2`	Сравнивает, равны ли итераторы (указывающие на один и тот же элемент)
`iterator-type{}`	Создает итератор через конструктор по умолчанию
`iterator-type{ itr }`	Создает итератор из `itr` через конструктор копирования
`itr1 = itr2`	Присваивает итератору `itr1` `itr2`

Двунаправленные итераторы

Двунаправленный итератор — это однонаправленный итератор, который также может выполнять итерацию в обратном направлении. Можно использовать двунаправленный итератор вместо однонаправленного и итератора ввода во всех случаях.

Двунаправленные итераторы допускают обратную итерацию с помощью `operator--` и `operator--(int)`. Контейнеры STL, которые обеспечивают двунаправленные итераторы, представляют собой `array`, `list`, `deque`, `vector` и все упорядоченные ассоциативные контейнеры.

Листинг 14.4 показывает, как выполнить итерацию в обоих направлениях, используя двунаправленный итератор в `list`.

Листинг 14.4. Методы std::list begin и end предоставляют двунаправленный итератор

```
#include <list>

TEST_CASE("std::list begin and end provide bidirectional iterators") {
  const std::list<int> easy_as{ 1, 2, 3 }; ❶
  auto itr = easy_as.begin(); ❷
  REQUIRE(*itr == 1); ❸
  itr++; ❹
  REQUIRE(*itr == 2);
  itr--; ❺
  REQUIRE(*itr == 1); ❻
  REQUIRE(itr == easy_as.cbegin());
}
```

Здесь создается `list`, содержащий три элемента ❶. Итератор с именем `itr` извлекается с помощью метода `begin` в `list` ❷. Как и в случае с итераторами `input` и `forward`, можно разыменовывать ❸ и увеличивать ❹ итератор. Кроме того, можно уменьшить итератор ❺, чтобы вернуться к элементам, которые были перебраны ❻.

Таблица 14.5 обобщает поддерживаемые операции двунаправленного итератора.

Таблица 14.5. Поддерживаемые операции двунаправленного итератора

Операция	Примечания
`*itr`	Разыменовывает член, на который направлен указатель. Может быть только для чтения (необязательно)
`itr->mbr`	Разыменовывает член `mbr` объекта, на который указывает `itr`
`++itr` `itr++`	Увеличивает итератор так, что тот указывает на следующий элемент
`--itr` `itr--`	Уменьшает итератор так, что тот указывает на предыдущий элемент
`itr1 == itr2` `itr1 != itr2`	Сравнивает, равны ли итераторы (указывающие на один и тот же элемент)
`iterator-type{}`	Создает итератор через конструктор по умолчанию
`iterator-type { itr }`	Создает итератор из `itr` через конструктор копирования
`itr1 = itr2`	Присваивает итератору `itr1` `itr2`

Итераторы с произвольным доступом

Итератор с произвольным доступом — это двунаправленный итератор, который поддерживает произвольный доступ к элементам. Можно использовать итератор с произвольным доступом вместо двунаправленных прямых итераторов и итераторов ввода во всех случаях.

Итераторы с произвольным доступом разрешают произвольный доступ с помощью `operator[]`, а также поддерживают арифметику итераторов, такую как сложение или вычитание целочисленных значений и вычитание других итераторов для нахождения расстояний. Контейнеры STL, которые предоставляют итераторы с произвольным доступом, — это `array`, `vector` и `deque`. В листинге 14.5 показано, как получить доступ к произвольным элементам, используя итератор произвольного доступа из вектора.

В коде создается `vector`, содержащий три элемента ❶. Итератор с именем `itr` извлекается с помощью метода `begin` в `vector` ❷. Поскольку это итератор с произвольным доступом, можно использовать `operator[]`для разыменования произвольных

элементов ❸. Конечно, все еще можно увеличить итератор, используя operator++ ❹. Также можно добавить или вычесть из итератора, чтобы получить доступ к элементам с заданным смещением ❺ ❻.

Листинг 14.5. Взаимодействие с итератором с произвольным доступом

```
#include <vector>

TEST_CASE("std::vector begin and end provide random-access iterators") {
  const std::vector<int> easy_as{ 1, 2, 3 }; ❶
  auto itr = easy_as.begin(); ❷
  REQUIRE(itr[0] == 1); ❸
  itr++; ❹
  REQUIRE(*(easy_as.cbegin() + 2) == 3); ❺
  REQUIRE(easy_as.cend() - itr == 2); ❻
}
```

Список поддерживаемых операций итератора с произвольным доступом

Таблица 14.6 обобщает поддерживаемые операции итератора с произвольным доступом.

Таблица 14.6. Поддерживаемые операции итератора с произвольным доступом

Операция	Примечания
`itr[n]`	Разыменовывает элемент с индексом `n`
`itr+n` `itr-n`	Возвращает итератор по смещению `n` от `itr`
`itr2-itr1`	Вычисляет расстояние между `itr1` и `itr2`
`*itr`	Разыменовывает член, на который направлен указатель. Может быть только для чтения (необязательно)
`itr->mbr`	Разыменовывает член `mbr` объекта, на который указывает `itr`
`++itr` `itr++`	Увеличивает итератор так, что тот указывает на следующий элемент
`--itr` `itr--`	Уменьшает итератор так, что тот указывает на предыдущий элемент
`itr1 == itr2` `itr1 != itr2`	Сравнивает, равны ли итераторы (указывающие на один и тот же элемент)
`iterator-type{}`	Создает итератор через конструктор по умолчанию
`iterator-type{ itr }`	Создает итератор из `itr` через конструктор копирования

`itr1 < itr2`
`itr1 > itr2`
`itr1 <= itr2`
`itr1 >= itr2`
	Выполняет соответствующее сравнение с позициями итераторов

Непрерывные итераторы

Непрерывный итератор — это итератор произвольного доступа с элементами, непрерывным в памяти. Для непрерывного итератора `itr` все элементы `itr[n]` и `itr[n + 1]` удовлетворяют следующему соотношению для всех допустимых выборок индексов `n` и смещений `i`:

```
&itr[n] + i == &itr[n+i]
```

Контейнеры `vector` и `array` предоставляют смежные итераторы, но `list` и `deque` этого не делают.

Изменяемые итераторы

Все однонаправленные итераторы, двунаправленные итераторы, итераторы с произвольным доступом и смежные итераторы могут поддерживать режимы только для чтения или чтения и записи. Если итератор поддерживает чтение и запись, можно присвоить значения ссылкам, возвращенным разыменованием итератора. Такие итераторы называются *изменяемыми итераторами*. Например, двунаправленный итератор, который поддерживает чтение и запись, называется изменяемым двунаправленным итератором.

До сих пор в каждом из примеров контейнеры, используемые для поддержки итераторов, были отмечены как `const`. Это создает итераторы для `const`-объектов, которые, конечно, не доступны для записи. В листинге 14.6 извлекается изменяемый итератор с произвольным доступом из (не-`const`) `deque`, что позволяет записывать в произвольные элементы контейнера.

Листинг 14.6. Изменяемый итератор с произвольным доступом разрешает запись

```
#include <deque>

TEST_CASE("Mutable random-access iterators support writing.") {
  std::deque<int> easy_as{ 1, 0, 3 }; ❶
  auto itr = easy_as.begin(); ❷
  itr[1] = 2; ❸
  itr++; ❹
  REQUIRE(*itr == 2); ❺
}
```

Создается `deque`, содержащая три элемента ❶, а затем получается итератор, указывающий на первый элемент ❷. Затем записывается значение 2 во второй элемент ❸. Затем итератор увеличивается, чтобы он указывал на элемент, который только что был изменен ❹. При разыменовывании указанного элемента возвращается значение, которое было записано в ❺.

На рис. 14.1 показаны отношения между итератором ввода и всеми его более характерными потомками.

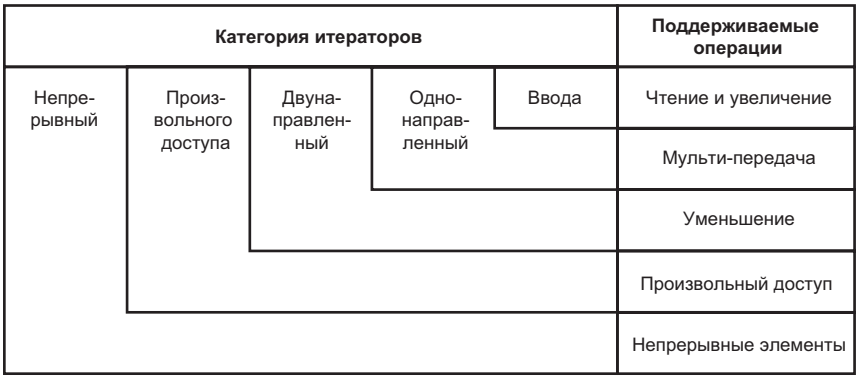


Рис. 14.1. Категории итераторов ввода и их вложенные отношения

Подводя итоги, итератор ввода поддерживает только чтение и увеличение. Однонаправленные итераторы также являются итераторами ввода, поэтому они также поддерживают чтение и увеличение, но дополнительно позволяют многократно выполнять итерации по их диапазону («многоходовой»). Двунаправленные итераторы также являются однонаправленными итераторами, но они дополнительно допускают операции уменьшения. Итераторы с произвольным доступом также являются двунаправленными, но можно напрямую обращаться к произвольным элементам в последовательности. И, наконец, непрерывные итераторы являются итераторами с произвольным доступом, которые гарантируют, что их элементы непрерывны в памяти.

Вспомогательные функции итераторов

Если вы пишете общий код, связанный с итераторами, стоит использовать *вспомогательные функции итераторов* из заголовка `<iterator>` для управления итераторами, а не для непосредственного использования поддерживаемых операций. Эти функции итераторов выполняют общие задачи перебора, обмена и вычисления расстояний между итераторами. Основное преимущество использования вспомогательных функций вместо прямого управления итератором заключается в том, что вспомогательные функции будут проверять характеристики типа итератора и определять наиболее эффективный метод для выполнения требуемой операции. Кроме того, вспомогательные функции итераторов делают общий код еще более универсальным, поскольку он будет работать с самым широким диапазоном итераторов.

std::advance

Вспомогательная функция итератора `std::advance` позволяет увеличивать или уменьшать значение на желаемую величину. Этот шаблон функции принимает

ссылку на итератор и целочисленное значение, соответствующее расстоянию, на которое нужно переместить итератор:

```
void std::advance(InputIterator&❶ itr, Distance❷ d);
```

Параметр шаблона `InputIterator` должен быть как минимум входным итератором ❶, а параметр шаблона `Distance` обычно является целым числом ❷.

Функция `advance` не выполняет проверку границ, поэтому нужно убедиться, что допустимый диапазон для позиции итератора не превышен.

В зависимости от категории итератора продвижение будет выполнять наиболее эффективную операцию, которая достигает желаемого эффекта.

- **Итератор ввода.** Функция `advance` будет вызывать `itr++` правильное количество раз; `dist` не может быть отрицательным.
- **Двунаправленный итератор.** Функция будет вызывать `itr++` или `itr--` правильное количество раз.
- **Итератор произвольного доступа.** Вызовет `itr + = dist`; `dist` может быть отрицательным.

ПРИМЕЧАНИЕ

Итераторы с произвольным доступом будут более эффективными, чем меньшие итераторы с `advance`, поэтому можно использовать `operator+=` вместо `advance`, если нужно запретить производительность в худшем случае (с линейным временем).

Листинг 14.7 показывает, как использовать `advance` для управления итератором с произвольным доступом.

Листинг 14.7. Использование `advance` для управления непрерывным итератором

```
#include <iterator>

TEST_CASE("advance modifies input iterators") {
  std::vector<unsigned char> mission{ ❶
    0x9e, 0xc4, 0xc1, 0x29,
    0x49, 0xa4, 0xf3, 0x14,
    0x74, 0xf2, 0x99, 0x05,
    0x8c, 0xe2, 0xb2, 0x2a
  };
  auto itr = mission.begin(); ❷
  std::advance(itr, 4); ❸
  REQUIRE(*itr == 0x49);
  std::advance(itr, 4); ❹
  REQUIRE(*itr == 0x74);
  std::advance(itr, -8); ❺
  REQUIRE(*itr == 0x9e);
}
```

Здесь инициализируется вектор под названием `mission` с 16 объектами `unsigned char` ❶. Затем извлекается итератор с именем `itr`, используя метод `begin` в `mission` ❷, и вызывается `advance` в `itr` для продвижения четырех элементов так, чтобы он указывал на четвертый элемент (со значением 0x49) ❸. Четыре элемента продвигаются к восьмому элементу (со значением 0x74) ❹. Наконец, вызывается `advance` с -8 для отступления на восемь значений, поэтому итератор снова указывает на первый элемент (со значением 0x9e) ❺.

std::next и std::prev

Вспомогательные функции итераторов `std::next` и `std::prev` являются шаблонами функций, которые вычисляют смещения от заданного итератора. Они возвращают новый итератор, указывающий на нужный элемент без изменения исходного итератора, как показано здесь:

```
ForwardIterator std::next(ForwardIterator& itr❶, Distance d=1❷);
BidirectionalIterator std::prev(BidirectionalIterator& itr❸, Distance d=1❹);
```

Затем функция принимает по меньшей мере однонаправленный итератор ❶ и, необязательно, расстояние ❷ и возвращает итератор, указывающий на соответствующее смещение. Это смещение может быть отрицательным, если `itr` является двунаправленным. Шаблон функции `prev` работает следующим образом в обратном порядке: он принимает по меньшей мере двунаправленный итератор ❸ и, необязательно, расстояние ❹ (которое может быть отрицательным).

Ни `next`, ни `prev` не выполняет проверку границ. Это означает, что вы должны быть полностью уверены в правильности вычислений и в том, что остаетесь в рамках последовательности; в противном случае вы получите неопределенное поведение.

ПРИМЕЧАНИЕ

И для `next`, и для `prev` `itr` остается неизменным, если только оно не является `r`-значением; в этом случае для эффективности используется `advance`.

Листинг 14.8 показывает, как использовать `next` для получения нового итератора, указывающего на элемент с заданным смещением.

Как и в листинге 14.7, инициализируется вектор, содержащий 16 `unsigned char`, и извлекается итератор `itr1`, указывающий на первый элемент ❶. `advance` используется, чтобы увеличить четыре элемента итератора ❷ так, чтобы он указывал на элемент со значением 0x49 ❸. Первое использование `next` опускает аргумент расстояния, который по умолчанию равен 1 ❹. Это создаст новый итератор, `itr2`, который равен `itr1` ❺. `next` вызывается во второй раз с аргументом расстояния со значением 4 ❻. Это создает еще один новый итератор, `itr3`, который указывает сдвиг на четыре после элемента `itr1` ❼. Ни один из этих вызовов не влияет на исходный итератор itr1 ❽.

Листинг 14.8. Использование next для получения смещений от итератора

```
#include <iterator>

TEST_CASE("next returns iterators at given offsets") {
  std::vector<unsigned char> mission{
    0x9e, 0xc4, 0xc1, 0x29,
    0x49, 0xa4, 0xf3, 0x14,
    0x74, 0xf2, 0x99, 0x05,
    0x8c, 0xe2, 0xb2, 0x2a
  };
  auto itr1 = mission.begin(); ❶
  std::advance(itr1, 4); ❷
  REQUIRE(*itr1 == 0x49); ❸

  auto itr2 = std::next(itr1); ❹
  REQUIRE(*itr2 == 0xa4); ❺

  auto itr3 = std::next(itr1, 4); ❻
  REQUIRE(*itr3 == 0x74); ❼

  REQUIRE(*itr1 == 0x49); ❽
}
```

std::distance

Вспомогательная функция итератора `std::distance` позволяет вычислить расстояние между двумя итераторами ввода, `itr1` и `itr2`:

```
Distance std::distance(InputIterator itr1, InputIterator itr2);
```

Если итераторы не имеют произвольного доступа, `itr2` должен ссылаться на элемент после `itr1`. Рекомендуется, чтобы `itr2` следовал за `itr1`, так как возникнет неопределенное поведение, если случайно нарушить это требование, и итераторы не будут иметь произвольный доступ.

Листинг 14.9 показывает, как вычислить расстояние между двумя итераторами с произвольным доступом.

Листинг 14.9. Использование расстояния для получения расстояния между итераторами

```
#include <iterator>

TEST_CASE("distance returns the number of elements between iterators") {
  std::vector<unsigned char> mission{ ❶
    0x9e, 0xc4, 0xc1, 0x29,
    0x49, 0xa4, 0xf3, 0x14,
    0x74, 0xf2, 0x99, 0x05,
    0x8c, 0xe2, 0xb2, 0x2a
  };
  auto eighth = std::next(mission.begin(), 8); ❷
  auto fifth = std::prev(eighth, 3); ❸
  REQUIRE(std::distance(fifth, eighth) == 3); ❹
}
```

После инициализации `vector` ❶ создается итератор, указывающий на элемент `eighth`, с использованием `std::next` ❷. `std::prev` используется для `eighth`, чтобы получить итератор для элемента `fifth`, передавая 3 в качестве второго аргумента ❸. При передаче `fifth` и `eighth` в качестве аргументов для расстояния получается 3 ❹.

std::iter_swap

Вспомогательная функция итератора `std::iter_swap` позволяет поменять местами значения, на которые указывают два однонаправленных итератора, `itr1` и `itr2`:

```
Distance std::iter_swap(ForwardIterator itr1, ForwardIterator itr2);
```

Итераторам не нужно иметь один и тот же тип, если типы, на которые они указывают, могут быть назначены друг другу. В листинге 14.10 показано, как использовать `iter_swap` для обмена двумя элементами `vector`.

Листинг 14.10. Использование `iter_swap` для обмена указанными элементами

```
#include <iterator>

TEST_CASE("iter_swap swaps pointed-to elements") {
  std::vector<long> easy_as{ 3, 2, 1 }; ❶
  std::iter_swap(easy_as.begin()❷, std::next(easy_as.begin(), 2)❸);
  REQUIRE(easy_as[0] == 1); ❹
  REQUIRE(easy_as[1] == 2);
  REQUIRE(easy_as[2] == 3);
}
```

После создания вектора с элементами `3 2 1` ❶ вызывается `iter_swap` для первого ❷ и последнего элементов ❸. После обмена вектор содержит элементы `1 2 3` ❹.

Дополнительные итераторные адаптеры

В дополнение к вставке итераторов STL предоставляет итераторные адаптеры переноса и обратные итераторные адаптеры для изменения поведения итератора.

ПРИМЕЧАНИЕ

`STL` также предоставляет адаптеры потоковых итераторов, о которых вы узнаете в главе 16 при изучении потоков.

Итераторные адаптеры переноса

Итераторный адаптер переноса — это шаблон класса, который преобразует все обращения итератора в операции переноса. Шаблон вспомогательной функции `std::make_move_iterator` в заголовке `<iterator>` принимает один аргумент итератора и возвращает адаптер итератора перемещения. Каноническое использование

адаптера итератора перемещения заключается в перемещении диапазона объектов в новый контейнер. Рассмотрим учебный класс `Movable` в листинге 14.11, в котором хранится значение `int` с именем `id`.

Листинг 14.11. Класс Movable хранит int

```
struct Movable{
  Movable(int id) : id{ id } { } ❶
  Movable(Movable&& m) {
    id = m.id; ❷
    m.id = -1; ❸
  }
  int id;
};
```

Конструктор `Movable` принимает `int` и сохраняет его в своем поле `id` ❶. `Movable` является еще и конструируемым; он украдет идентификатор из аргумента конструктора переноса ❷, заменив его на -1 ❸.

Листинг 14.12 создает `vectorMovable` объектов, называемый `donor`, и перемещает их в `vector`, называемый `recipient`.

Здесь по умолчанию создается `vector` с именем `donor` ❶, который используется для вызова `emplace_back` для трех объектов `Movable` с полями `id` 1, 2 и 3 ❷. Затем используется конструктор диапазона `vector` с итераторами `donorbegin` и `end`, который передается в `make_move_iterator` ❸. Это преобразует все операции итератора в операции переноса, поэтому вызывается конструктор переноса в `Movable`. В результате все элементы `donor` находятся в состоянии перемещения ❹, и все элементы `recipient` соответствуют предыдущим элементам `donor` ❺.

Листинг 14.12. Использование итераторного адаптера переноса для преобразования операций итератора в операции переноса

```
#include <iterator>

TEST_CASE("move iterators convert accesses into move operations") {
  std::vector<Movable> donor; ❶
  donor.emplace_back(1); ❷
  donor.emplace_back(2);
  donor.emplace_back(3);
  std::vector<Movable> recipient{
    std::make_move_iterator(donor.begin()), ❸
    std::make_move_iterator(donor.end()),
  };
  REQUIRE(donor[0].id == -1); ❹
  REQUIRE(donor[1].id == -1);
  REQUIRE(donor[2].id == -1);
  REQUIRE(recipient[0].id == 1); ❺
  REQUIRE(recipient[1].id == 2);
  REQUIRE(recipient[2].id == 3);
}
```

Обратные итераторные адаптеры

Обратный итераторный адаптер — это шаблон класса, который меняет операторы увеличения и уменьшения итератора. Чистым эффектом является то, что можно обратить входные данные к алгоритму, применив обратный итераторный адаптер. Один из распространенных сценариев, в котором можно использовать обратный итератор, — это поиск в обратном направлении от конца контейнера. Например, возможно, вы помещаете журналы в конец `deque` и хотите найти самую последнюю запись, которая соответствует некоторому критерию.

Почти все контейнеры в главе 13 предоставляют обратные итераторы с помощью методов `rbegin/rend/crbegin/crend`. Например, можно создать контейнер с обратной последовательностью другого контейнера, как показано в листинге 14.13.

Листинг 14.13. Создание контейнера с обращением элементов другого контейнера

```
TEST_CASE("reverse iterators can initialize containers") {
  std::list<int> original{ 3, 2, 1 }; ❶
  std::vector<int> easy_as{ original.crbegin(), original.crend() }; ❷
  REQUIRE(easy_as[0] == 1); ❸
  REQUIRE(easy_as[1] == 2);
  REQUIRE(easy_as[2] == 3);
}
```

Здесь создается `list`, содержащий элементы `3 2 1` ❶. Затем создается `vector` с обратной последовательностью с использованием методов `crbegin` и `crend` ❷. `vector` содержит 1 2 3, то есть элементы списка в обратном порядке ❸.

Хотя контейнеры обычно предоставляют однонаправленные итераторы напрямую, также можно вручную преобразовать обычный итератор в обратный. Шаблон вспомогательной функции `std::make_reverse_iterator` в заголовке `<iterator>` принимает один аргумент итератора и возвращает обратный итераторный адаптер.

Обратные итераторы предназначены для работы с полуоткрытыми диапазонами, которые полностью противоположны нормальным полуоткрытым диапазонам. Под капотом *обратный полуоткрытый диапазон* имеет итератор `rbegin`, который ссылается на 1 после конца полуоткрытого диапазона, и итератор `rend`, который ссылается на начало полуоткрытого диапазона, как показано на рис. 14.2.

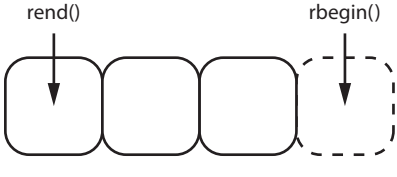


Рис. 14.2. Обратный полуоткрытый диапазон

Однако все эти детали реализации прозрачны для пользователя. Итераторы разыменовываются, как и следовало ожидать. Пока диапазон не пуст, можно разыменовать

обратный итератор начала, и он вернет первый элемент. Однако *нельзя* разыменовать обратный итератор конца.

Зачем вводить это усложнение? С этим дизайном можно легко поменять местами начальный и конечный итераторы полуоткрытого диапазона, чтобы получить обратный полуоткрытый диапазон. Например, в листинге 14.14 используется `std::make_reverse_iterator` для преобразования обычных итераторов в обратные итераторы, выполняя ту же задачу, что и в листинге 14.13.

Листинг 14.14. Функция make_reverse_iterator преобразует обычный итератор в обратный

```
TEST_CASE("make_reverse_iterator converts a normal iterator") {
  std::list<int> original{ 3, 2, 1 };
  auto begin = std::make_reverse_iterator(original.cend()); ❶
  auto end = std::make_reverse_iterator(original.cbegin()); ❷
  std::vector<int> easy_as{ begin, end }; ❸
  REQUIRE(easy_as[0] == 1);
  REQUIRE(easy_as[1] == 2);
  REQUIRE(easy_as[2] == 3);
}
```

Обратите особое внимание на итераторы, которые извлекаются из `original`. Чтобы создать итератор `begin`, извлекается итератор `end` из `original` и передается в `make_reverse_iterator` ❶. Адаптер обратного итератора поменяет местами операторы увеличения и уменьшения, но его нужно запустить в нужном месте. Аналогично нужно завершить выполнение в начале оригинала, поэтому результат `cbegin` передается в `make_reverse_iterator`, чтобы получить правильное завершение ❷. Передача их в конструктор диапазона `easy_as` ❸ приводит к результатам, идентичным листингу 14.13.

ПРИМЕЧАНИЕ

Все обратные итераторы предоставляют метод `base`, который преобразует обратный итератор обратно в обычный итератор.

Итоги

В этой короткой главе вы узнали все категории итераторов: вывода, ввода, однонаправленный, двунаправленный, с произвольным доступом и непрерывный. Знание основных свойств каждой категории предоставляет основу для понимания того, как контейнеры соединяются с алгоритмами. В этой главе также были рассмотрены итераторные адаптеры, которые позволяют настраивать поведение итераторов, и вспомогательные функции итераторов, которые помогают писать общий код с итераторами.

Упражнения

14.1. Создайте следствие для листинга 14.8, используя `std::prev` вместо `std::next`.

14.2. Напишите шаблон функции с именем `sum`, который принимает полуоткрытый диапазон объектов `int` и возвращает сумму последовательности.

14.3. Напишите программу, которая использует класс `Stopwatch` в листинге 12.25 для определения производительности `std::advance` во время выполнения при получении прямого итератора из большого `std::forward_list` и большого `std::vector`. Как время выполнения меняется с количеством элементов в контейнере? (Попробуйте сотни тысяч или миллионы элементов.)

Что еще почитать?

- «Стандартная библиотека C++. Справочное руководство», 2-е издание, Николай М. Джосаттис (Вильямс, 2017)
- «Шаблоны C++. Справочник разработчика», 2-е издание, Дэвид Вандевурд (Вильямс, 2018)

15 Строки

Если вы разговариваете с человеком на языке, который он понимает, вы обращаетесь к его разуму. Если вы разговариваете с ним на его языке, то обращаетесь к его сердцу.

Нельсон Мандела

STL предоставляет специальный *строковый контейнер* для данных на естественном языке, таких как слова, предложения и языки разметки. Доступный в заголовке `<string>`, `std::basic_string` — это шаблон класса, который можно специализировать на базовом символьном типе строки. Как последовательный контейнер `basic_string`, по сути, похож на вектор, но с некоторыми специальными возможностями для манипулирования языком.

`basic_string` в STL обеспечивает значительные улучшения безопасности и функциональности по сравнению со строками в стиле C или строками с нулевыми символами в конце, и поскольку данные на человеческом языке затопляют большинство современных программ, вы, вероятно, найдете `basic_string` незаменимым.

std::string

STL предоставляет четыре специализации `basic_string` в заголовке `<string>`.

Каждая специализация реализует строку с использованием одного из основных типов символов, о которых вы узнали в главе 2:

- `std::string` для `char` используется для кодировок, таких как ASCII;
- `std::wstring` для `wchar_t` достаточно большой, чтобы содержать самый большой символ языкового стандарта реализации;

- `std::u16string` для `char16_t` используется для кодировок, таких как UTF-16;
- `std::u32string` для `char32_t` используется для кодировок, таких как UTF-32.

Вы будете использовать специализацию с соответствующим базовым типом. Поскольку эти специализации имеют одинаковый интерфейс, все примеры в этой главе будут использовать `std::string`.

Создание

Контейнер `basic_string` принимает три параметра шаблона:

- базовый тип символа, `T`;
- типажи базового типа, `Traits`;
- распределитель, `Alloc`.

Из них обязательным является только `T`. Класс шаблона STL `std::char_traits` в заголовке `<string>` абстрагирует символьные и строковые операции от базового символьного типа. Кроме того, если вы не планируете поддерживать пользовательский тип символов, не нужно будет реализовывать собственные типажи типа, потому что у `char_traits` есть специализации, доступные для `char`, `wchar_t`, `char16_t` и `char32_t`. Когда stdlib предоставляет специализации для типа, не нужно предоставлять его самостоятельно, если только не требуется какое-то экзотическое поведение.

Вместе специализация `basic_string` выглядит следующим образом, где `T` — тип символа:

```
std::basic_string<T, Traits=std::char_traits<T>, Alloc=std::allocator<T>>
```

ПРИМЕЧАНИЕ

В большинстве случаев вам придется иметь дело с одной из предопределенных специализаций, особенно с `string` или `wstring`. Однако если нужен пользовательский распределитель, придется соответствующим образом специализировать `basic_string`.

Контейнер `basic_string<T>` поддерживает те же конструкторы, что и `vector <T>`, плюс дополнительные удобные конструкторы для преобразования строки в стиле C. Другими словами, `string` поддерживает конструкторы `vector<char>`, строка `wstring` поддерживает конструкторы `vector<wchar_t>` и т. д. Как и в случае с `vector`, используйте скобки для всех конструкторов `basic_string`, кроме случаев, когда действительно нужен список инициализаторов.

Можно по умолчанию создать пустую строку или, если нужно заполнить строку повторяющимся символом, используйте конструктор заполнения, передавая `size_t` и `char`, как показано в листинге 15.1.

После того как вы по умолчанию создаете `string` ❶, она не содержит элементов ❷. Если нужно заполнить строку повторяющимися символами, используйте кон-

структор заполнения, передав число элементов, которые нужно заполнить, и их значение ❸. Пример заполняет строку тремя символами A ❹.

Листинг 15.1. Конструкторы по умолчанию и заполнители строки

```
#include <string>
TEST_CASE("std::string supports constructing") {
  SECTION("empty strings") {
    std::string cheese; ❶
    REQUIRE(cheese.empty()); ❷
  }
  SECTION("repeated characters") {
    std::string roadside_assistance(3, 'A'); ❸
    REQUIRE(roadside_assistance == "AAA"); ❹
  }
}
```

ПРИМЕЧАНИЕ

Позже в этой главе вы узнаете о сравнении `std::string` с помощью `operator==` . Поскольку строки в стиле `C`, как правило, обрабатываются с обычными указателями или массивами, `operator==` возвращает `true` только при наличии одного и того же объекта. Однако для `std::string` `operator==` возвращает `true`, если содержимое эквивалентно. Как видно из листинга 15.1, сравнение работает, даже если один из операндов является строковым литералом в стиле `C`.

Конструктор `string` также предлагает два конструктора на основе `const char*`. Если аргумент указывает на строку с нулевым символом в конце, конструктор `string` может самостоятельно определить длину ввода. Если указатель не указывает на строку с нулевым символом в конце или если нужно использовать только первую часть строки, можно передать аргумент длины, который сообщает конструктору `string`, сколько элементов скопировать, как показано в листинге 15.2.

Листинг 15.2. Построение string из строк в стиле C

```
TEST_CASE("std::string supports constructing substrings ") {
  auto word = "gobbledygook"; ❶
  REQUIRE(std::string(word) == "gobbledygook"); ❷
  REQUIRE(std::string(word, 6) == "gobble"); ❸
}
```

В примере создается `const char*` с именем `word`, указывающим на строковый литерал `gobbledygook` ❶ в стиле C. Затем создается `string`, передавая `word`. Как и ожидалось, результирующая `string` содержит `gobbledygook` ❷. В следующем тесте передается число 6 в качестве второго аргумента. Это заставляет `string` принимать только первые шесть символов слова, в результате чего получается `string`, содержащая `gobble` ❸.

Кроме того, можно создавать `string` из других строк. Как контейнер STL, строка полностью поддерживает семантику копирования и переноса. Также можно создать `string` из *подстроки* — непрерывного подмножества другой строки. В листинге 15.3 показаны эти три конструктора.

Листинг 15.3. Копирование, перенос и создание подстроки объектов string

```
TEST_CASE("std::string supports") {
  std::string word("catawampus"); ❶
  SECTION("copy constructing") {
    REQUIRE(std::string(word) == "catawampus"); ❷
  }
  SECTION("move constructing") {
    REQUIRE(std::string(move(word)) == "catawampus"); ❸
  }
  SECTION("constructing from substrings") {
    REQUIRE(std::string(word, 0, 3) == "cat"); ❹
    REQUIRE(std::string(word, 4) == "wampus"); ❺
  }
}
```

ПРИМЕЧАНИЕ

В листинге 15.3 `word` находится в состоянии перемещения, о котором вы помните из раздела «Семантика перемещения» на с. 184, оно означает, что `word` может быть только переназначен или уничтожен.

Здесь создается строка под названием `word`, содержащая символы `catawampus` ❶. Конструктор копирования приводит к следующему листингу 15.4: создание строки, содержащей копию символов `word` ❷. Конструктор переноса крадет символы `word`, в результате чего получается новая строка, содержащая `catawampus` ❸. Наконец, можно создать новую `string` на основе подстрок. Путем передачи `word`, начальной позиции 0 и длины 3 создается новая `string`, содержащая символы `cat` ❹. Если вместо этого передать слово и начальную позицию 4 (без длины), вы получите все символы от четвертого до конца исходной строки, что приведет к значению `wampus` ❺.

Класс `string` также поддерживает создание литералов с помощью `std::string_literals::operator""s`. Основным преимуществом является удобство обозначений, но также можно использовать `operator""s`, чтобы легко вставлять нулевые символы в строку, как показано в листинге 15.4.

Листинг 15.4. Создание строки

```
TEST_CASE("constructing a string with") {
  SECTION("std::string(char*) stops at embedded nulls") {
    std::string str("idioglossia\0ellohay!"); ❶
    REQUIRE(str.length() == 11); ❷
  }
  SECTION("operator\"\"s incorporates embedded nulls") {
    using namespace std::string_literals; ❸
    auto str_lit = "idioglossia\0ellohay!"s; ❹
    REQUIRE(str_lit.length() == 20); ❺
  }
}
```

В первом тесте создается `string` при помощи литерала `idioglossia\0ellohay!` ❶, что приводит к `string`, содержащей `idioglossia` ❷. Остальная часть литерала не была скопирована в `string` из-за встроенных нулей. Во втором тесте вводится пространство имен `std::string_literals` ❸, чтобы можно было использовать `operator""s` для создания `string` непосредственно из литерала ❹. В отличие от конструктора `std::string` ❶, `operator""s` выдает строку, содержащую весь литерал — встроенные нулевые байты и все остальное ❺.

Таблица 15.1 обобщает варианты построения строки. В этой таблице `c` означает `char`, `n` и `pos` — `size_t`, `str` — `string` или строку в стиле C, `c_str` — строку в стиле C, а `beg` и `end` — итераторы ввода.

Таблица 15.1. Поддерживаемые конструкторы `std::string`

Конструктор	Создает `string`, содержащую ...
`string()`	Ноль символов
`string(n, c)`	`c`, повторенный `n` раз
`string(str, pos, [n])`	Полуоткрытый диапазон от `pos` до `pos+n` от `str`. Подстрока расширяет `rom pos` до конца `str`, если `n` опущен
`string(c_str, [n])`	Копию `c_str` длиной `n`. Если `c_str` заканчивается нулем, `n` по умолчанию равен длине строки с нулевым символом в конце
`string(beg, end)`	Копию элементов в полуоткрытом диапазоне от `beg` до `end`
`string(str)`	Копию `str`
`string(move(str))`	Содержимое `str`, которое находится в перемещенном состоянии после создания
`string{ c1, c2, c3 }`	Символы `c1`, `c2` и `c3`
`"my string literal"s`	Строку, содержащую символы `my string literal`

Хранение строк и оптимизация небольших строк

Точно так же, как `vector`, `string` использует динамическое хранилище для непрерывного хранения составляющих ее элементов. Соответственно, вектор и строка имеют очень похожую семантику копирования/переноса-создания/присваивания. Например, операции копирования потенциально более затратны, чем операции переноса, поскольку содержащиеся в них элементы находятся в динамической памяти.

Самые популярные реализации STL имеют *оптимизации небольших строк* (small string optimizations, SSO). SSO помещает содержимое строки в хранилище объекта (а не в динамическое хранилище), если содержимое достаточно мало. Как правило, кандидатом для SSO является `string` длиной менее 24 байт. Реализаторы предостав-

ляют эту оптимизацию, потому что во многих современных программах большинство строк `string`. (У `vector` нет оптимизаций небольших строк.)

ПРИМЕЧАНИЕ

Практически `SSO` влияет на перенос двумя способами. Во-первых, любые ссылки на элементы `string` будут недействительными, если строка перемещается. Во-вторых, переносы для `string` могут быть медленнее, чем для `vector`, потому что `string` должны проверять наличие `SSO`.

`string` имеет *размер* (или *длину*) и *емкость*. Размер — это количество символов, содержащихся в строке, а емкость — это количество символов, которое строка может хранить без изменения размера.

Таблица 15.2 содержит методы для чтения и управления размером и емкостью `string`. В этой таблице `n` является `size_t`. Звездочка (`*`) указывает на то, что эта операция делает недействительными обычные указатели и итераторы для элементов `s`, по крайней мере в некоторых случаях.

Таблица 15.2. Поддерживаемые методы хранения и длины `std::string`

Метод	Возвращаемое значение
`s.empty()`	`true`, если `s` не содержит символов; иначе — `false`
`s.size()`	Количество символов в `s`
`s.length()`	Идентично `s.size()`
`s.max_size()`	Максимально возможный размер `s` (из-за ограничений системы/времени выполнения)
`s.capacity()`	Количество символов `s`, которое может быть сохранено до изменения размера
`s.shrink_to_fit()`	`void`; выдает необязательный запрос на уменьшение `s.capacity()` до `s.size()` *
`s.reserve([n])`	`void`; если `n> s.capacity()`, изменяет размеры, так чтобы `s` могла содержать не менее `n` элементов; в противном случае выдает необязательный `request*`, чтобы уменьшить `s.capacity()` до `n` или `s.size()` в зависимости от того, какое значение больше

ПРИМЕЧАНИЕ

На момент выхода книги черновой стандарт C++ 20 изменял поведение метода `reserve`, когда его аргумент меньше размера `string`. Оно будет соответствовать поведению `vector`, то есть отсутствию эффекта, а не будет эквивалентно вызову `shrink_to_fit`.

Обратите внимание, что методы размера и емкости `string` очень похожи на методы `vector`. Это прямой результат близости их моделей хранения.

Поэлементный и итераторный доступ

Поскольку `string` предлагает итераторы с произвольным доступом к непрерывным элементам, она, соответственно, предоставляет аналогичные методы поэлементного и итераторного доступа, как и `vector`.

Для взаимодействия с API в стиле C `string` также предоставляет метод `c_str`, который возвращает неизменяемую версию строки с нулевым символом в конце как `const char*`, что показано в листинге 15.5.

Листинг 15.5. Извлечение строки с нулевым символом в конце

```
TEST_CASE("string's c_str method makes null-terminated strings") {
  std::string word("horripilation"); ❶
  auto as_cstr = word.c_str(); ❷
  REQUIRE(as_cstr[0] == 'h'); ❸
  REQUIRE(as_cstr[1] == 'o');
  REQUIRE(as_cstr[11] == 'o');
  REQUIRE(as_cstr[12] == 'n');
  REQUIRE(as_cstr[13] == '\0'); ❹
}
```

Создается строка с именем `word`, содержащая символы `horripilation` ❶, и используется метод `c_str` для извлечения строки с нулевым символом в конце под названием `as_cstr` ❷. Поскольку `as_cstr` является `const char*`, можно использовать `operator []`, чтобы показать, что он содержит те же символы, что и `word` ❸, и что имеет нулевое окончание ❹.

ПРИМЕЧАНИЕ

Класс `std::string` поддерживает `operator[]`, который ведет себя так же, как и строка в стиле C.

Как правило, `c_str` и `data` дают идентичные результаты, за исключением того, что ссылки, возвращаемые `data`, могут быть не-`const`. Всякий раз при манипуляциях со `string` реализации обычно гарантируют, что непрерывная память, поддерживающая `string`, заканчивается нулевым символом в конце строки. Программа из листинга 15.6 показывает это поведение, выводя результаты вызова `data` и `c_str` вместе с их адресами.

Листинг 15.6. Пример того, что c_str и data возвращают эквивалентные адреса

```
#include <string>
#include <cstdio>

int main() {
  std::string word("pulchritudinous");
  printf("c_str: %s at 0x%p\n", word.c_str(), word.c_str()); ❶
  printf("data:  %s at 0x%p\n", word.data(), word.data()); ❷
}
--------------------------------------------------------------------------------
```

```
c_str: pulchritudinous at 0x0000002FAE6FF8D0 ❶
data:  pulchritudinous at 0x0000002FAE6FF8D0 ❷
```

И `c_str`, и `data` дают одинаковые результаты, так как указывают на одни и те же адреса ❶ ❷. Поскольку адрес является началом `string` с нулевым значением, `printf` выдает одинаковые выходные данные для обоих вызовов.

В таблице 15.3 перечислены методы доступа к строке. Обратите внимание, что `n` означает `size_t` в таблице.

Таблица 15.3. Поддерживаемые методы поэлементного и итераторного доступа в `std::string`

Метод	Возвращаемое значение
`s.begin()`	Итератор, указывающий на первый элемент
`s.cbegin()`	Итератор `const`, указывающий на первый элемент
`s.end()`	Итератор, указывающий на элемент, следующий за последним
`s.cend()`	Итератор `const`, указывающий на элемент, следующий за последним
`s.at(n)`	Ссылка на элемент `n` из `s`. Возвращает `std::out_of_range`, если выходит за пределы строки
`s[n]`	Ссылка на элемент `n` из `s`. Возвращает неопределенное поведение, если `n>s.size()`. Также `s[s.size()]` должно быть 0, поэтому запись ненулевого значения в этот символ приведет к неопределенному поведению
`s.front()`	Ссылка на первый элемент
`s.back()`	Ссылка на последний элемент
`s.data()`	Обычный указатель на первый элемент, если строка не пустая. Для пустой строки возвращает указатель на нулевой символ
`s.c_str()`	Возвращает неизменяемую, завершенную нулем версию содержимого `s`

Сравнение строк

Обратите внимание, что строка поддерживает сравнение с другими строками и с обычными строками в стиле C при использовании обычных операторов сравнения. Например, равенство `operator==` возвращает `true`, если размер и содержимое левой и правой части равны, а неравенство `operator!=` возвращает противоположное значение. Остальные операторы сравнения выполняют *лексикографическое сравнение*, то есть сортируют по алфавиту, где $A < Z < a < z$ и где, если все остальное равно, более короткие слова меньше, чем более длинные слова (например, *pal* < *palindrome*). В листинге 15.7 даны такие сравнения.

Здесь вводится пространство имен `std::literals::string_literals`, чтобы можно было легко создать строку с помощью `operator""s` ❶. Также создается `string` под

названием `word`, содержащая символы `allusion` ❷. В первом наборе тестов изучаются `operator==` и `operator!=`.

Листинг 15.7. Класс string поддерживает сравнение

```
TEST_CASE("std::string supports comparison with") {
  using namespace std::literals::string_literals; ❶
  std::string word("allusion"); ❷
  SECTION("operator== and !=") {
    REQUIRE(word == "allusion"); ❸
    REQUIRE(word == "allusion"s); ❹
    REQUIRE(word != "Allusion"s); ❺
    REQUIRE(word != "illusion"s); ❻
    REQUIRE_FALSE(word == "illusion"s); ❼
  }
  SECTION("operator<") {
    REQUIRE(word < "illusion"); ❽
    REQUIRE(word < "illusion"s); ❾
    REQUIRE(word > "Illusion"s); ❿
  }
}
```

Вы можете заметить, что `word` равна (`==`) `allusion` как строка в стиле C ❸ и `string` ❹, но она не равна (`!=`) `string`, содержащим `Allusion` ❺ или `illusion` ❻. Как обычно, `operator==` и `operator!=` всегда возвращают противоположные результаты ❼.

Следующий набор тестов использует `operator<`, чтобы показать, что `allusion` меньше `illusion` ❽, потому что *a* лексикографически меньше, чем *i*. Сравнение работает со `string` и строками в стиле C ❾. Листинг 15.7 также показывает, что `Allusion` меньше, чем `allusion` ❿, потому что *A* лексикографически меньше, чем *a*.

Таблица 15.4 перечисляет методы сравнения `string`. Обратите внимание, что `other` — это `string` или строка типа `char*` в стиле C в таблице.

Таблица 15.4. Поддерживаемые операторы сравнения `std::string`

Метод	Возвращаемое значение
`s == other`	`true`, если `s` и другие имеют одинаковые символы и длины; в противном случае `false`
`s != other`	Противоположность оператору `==`
`s.compare(other)`	Возвращает `0`, если `s == other`, отрицательное число, если `s < other`, и положительное число, если `s > other`

`s < other`
`s > other`
`s <= other`
`s >= other`
	Результат соответствующей операции сравнения, согласно лексикографическому порядку

ПРИМЕЧАНИЕ

Технически лексикографическое сравнение зависит от кодировки строки. Теоретически возможно, что система может использовать кодировку по умолчанию, где алфавит находится в некотором беспорядочном состоянии (например, почти устаревшая кодировка `EBCDIC`, которая помещает строчные буквы перед прописными), что может повлиять на сравнение строк. По поводу `ASCII`-совместимых кодировок не нужно беспокоиться, поскольку они подразумевают ожидаемое лексикографическое поведение.

Управление элементами

`string` имеет *множество* методов для работы с элементами. Она поддерживает все методы `vector<char>`, а также многие другие, полезные для манипулирования языковыми данными.

Добавление элементов

Чтобы добавить элементы в строку, можно использовать `push_back`, который вставляет один символ в конце. Если нужно вставить более одного символа в конец `string`, используйте `operator+=` для добавления символа, строки `char*` с нулевым символом в конце или `string`. Также можно использовать метод `append`, который имеет три перегрузки. Во-первых, можно передать `string` или строку`char*` с нулевым символом в конце, необязательное смещение в этой строке и необязательное количество символов для добавления. Во-вторых, можно передать длину и `char`, который добавит это число символов в строку. В-третьих, можно добавить полуоткрытый диапазон. В листинге 15.8 показаны все эти операции.

Листинг 15.8. Добавление к string

```
TEST_CASE("std::string supports appending with") {
  std::string word("butt"); ❶
  SECTION("push_back") {
    word.push_back('e'); ❷
    REQUIRE(word == "butte");
  }
  SECTION("operator+=") {
    word += "erfinger"; ❸
    REQUIRE(word == "butterfinger");
  }
  SECTION("append char") {
    word.append(1, 's'); ❹
    REQUIRE(word == "butts");
  }
  SECTION("append char*") {
    word.append("stockings", 5); ❺
    REQUIRE(word == "buttstock");
  }
  SECTION("append (half-open range)") {
    std::string other("onomatopoeia"); ❻
    word.append(other.begin(), other.begin()+2); ❼
    REQUIRE(word == "button");
  }
}
```

Для начала инициализируется строка `word`, содержащая символы `butt` ❶. В первом тесте вызывается `push_back` с буквой `e` ❷, который возвращает `butte`. Затем добавляется `erfinger` к `word` с использованием `operator+=` ❸, получаем `butterfinger`. В первом вызове `append` добавляется один `s` ❹ для получения `butts`. (Эта настройка работает так же, как `push_back`.) Вторая перегрузка `append` позволяет указать `char*` и длину. Предоставляя `stockings` и длину 5, вы добавляете `stock` к `word`, чтобы получить `buttstock` ❺. Поскольку `append` работает с полуоткрытыми диапазонами, также можно создать строку с именем `other`, содержащую символы `onomatopoeia` ❻, и добавить первые два символа через полуоткрытый диапазон, чтобы получить `button` ❼.

ПРИМЕЧАНИЕ

Вспомните раздел «Тестовые случаи и разделы» на с. 379, где говорилось, что каждая `SECTION` юнит-теста `Catch` выполняется независимо, поэтому модификации `word` не зависят друг от друга: код настройки сбрасывает `word` для каждого теста.

Удаление элементов

Для удаления элементов из `string` есть несколько вариантов. Самый простой способ — использовать `pop_back`, который следует за `vector` при удалении последнего символа из `string`. Если вместо этого нужно удалить все символы (чтобы получить пустую `string`), используйте метод `clear`. Когда нужно больше точности в удалении элементов, используйте метод `erase`, который обеспечивает несколько перегрузок. Можно предоставить индекс и длину, что удаляет соответствующие символы. Также можно предоставить итератор для удаления одного элемента или полуоткрытого диапазона для удаления нескольких. Листинг 15.9 показывает удаление элементов из строки.

Листинг 15.9. Удаление элементов из string

```
TEST_CASE("std::string supports removal with") {
  std::string word("therein"); ❶
  SECTION("pop_back") {
    word.pop_back();
    word.pop_back(); ❷
    REQUIRE(word == "there");
  }
  SECTION("clear") {
    word.clear(); ❸
    REQUIRE(word.empty());
  }
  SECTION("erase using half-open range") {
    word.erase(word.begin(), word.begin()+3); ❹
    REQUIRE(word == "rein");
  }
  SECTION("erase using an index and length") {
    word.erase(5, 2);
    REQUIRE(word == "there"); ❺
  }
}
```

Создается строка `word`, содержащая `therein` ❶. В первом тесте дважды вызывается `pop_back`, чтобы сначала удалить букву `n`, за которой следует буква `i`, так что `word` содержит `there` ❷. Далее вызывается команда `clear`, которая удаляет все символы из `word`, поэтому `string` становится `empty` ❸. Последние два теста используют `erase`, чтобы удалить некоторое подмножество символов в `word`. При первом использовании удаляются первые три символа с полуоткрытым диапазоном, чтобы `word` содержала `rein` ❹. Во втором случае удаляются символы, начиная с индекса 5 (`i` в `therein`), и длиной в два символа ❺. Как и в первом тесте, получаются символы `there`.

Замена элементов

Чтобы вставить и удалить элементы одновременно, используйте `string` для предоставления метода `replace`, который имеет много перегрузок.

Во-первых, можно указать полуоткрытый диапазон и `char*` с нулевым символом в конце или `string`, а `replace` выполнит одновременный `erase` всех элементов в пределах полуоткрытого диапазона и `insert` предоставленной `string`, где использовался диапазон. Во-вторых, можно указать два полуоткрытых диапазона, и `replace` вставит второй диапазон вместо `string`.

Вместо замены диапазона можно использовать либо индекс, либо один итератор и длину. Можно указать новый полуоткрытый диапазон, символ и размер или `string`, и `replace` заменит новые элементы в пределах подразумеваемого диапазона. В листинге 15.10 показаны некоторые из этих возможностей.

Листинг 15.10. Замена элементов строки

```
TEST_CASE("std::string replace works with") {
  std::string word("substitution"); ❶
  SECTION("a range and a char*") {
    word.replace(word.begin()+9, word.end(), "e"); ❷
    REQUIRE(word == "substitute");
  }
  SECTION("two ranges") {
    std::string other("innuendo");
    word.replace(word.begin(), word.begin()+3,
                 other.begin(), other.begin()+2); ❸
    REQUIRE(word == "institution");
  }
  SECTION("an index/length and a string") {
    std::string other("vers");
    word.replace(3, 6, other); ❹
    REQUIRE(word == "subversion");
  }
}
```

Здесь создается строка `word`, содержащая `substitution` ❶. В первом тесте заменяются все символы от индекса 9 до конца буквой `e`, в результате чего значение `word` становится равным `substitute` ❷. Затем заменяются первые три буквы `word` пер-

выми двумя буквами `string`, содержащей `innuendo` ❸, в результате чего получается `institution`. Наконец, используется альтернативный способ задания целевой последовательности с индексом и длиной, чтобы заменить символы `stitut` на символы `vers`, что приводит к значению `subversion` ❹.

Класс `string` предлагает метод `resize` для ручной установки длины строки. Метод `resize` принимает два аргумента: новую длину и необязательный `char`. Если новая длина строки меньше, при изменении размера `char` игнорируется. Если новая длина `string` больше, функция `resize` добавляет `char` нужное количество раз для достижения желаемой длины. В листинге 15.11 показан метод `resize`.

Листинг 15.11. Изменение размера string

```
TEST_CASE("std::string resize") {
  std::string word("shamp"); ❶
  SECTION("can remove elements") {
    word.resize(4); ❷
    REQUIRE(word == "sham");
  }
  SECTION("can add elements") {
    word.resize(7, 'o'); ❸
    REQUIRE(word == "shampoo");
  }
}
```

Создается строка `word`, содержащая символы `shamp` ❶. В первом тесте изменяется размер `word` до длины 4, чтобы оно содержало `sham` ❷. Во втором тесте изменяется размер до длины 7 и предоставляется необязательный символ `o` в качестве значения для расширения `word` ❸. Это приводит к `word`, содержащему `shampoo`.

В разделе «Создание» на с. 415 объясняется конструктор подстроки, который может извлекать непрерывные последовательности символов для создания новой строки. Можно генерировать подстроки, используя метод `substr`, который принимает два необязательных аргумента: аргумент позиции и длину. Положение по умолчанию равно 0 (начало строки), а длина по умолчанию равна оставшейся части строки. В листинге 15.12 показано, как использовать `substr`.

Листинг 15.12. Извлечение подстрок из string

```
TEST_CASE("std::string substr with") {
  std::string word("hobbits"); ❶
  SECTION("no arguments copies the string") {
    REQUIRE(word.substr() == "hobbits"); ❷
  }
  SECTION("position takes the remainder") {
    REQUIRE(word.substr(3) == "bits"); ❸
  }
  SECTION("position/index takes a substring") {
    REQUIRE(word.substr(3, 3) == "bit"); ❹
  }
}
```

Объявляется `string` с именем `word`, содержащая `hobbits` ❶. Если вызвать `substr` без аргументов, происходит простое копирование `string` ❷. При предоставлении аргумента позиции 3 `substr` извлекает подстроку, начинающуюся с элемента 3 и продолжающуюся до конца `string`, возвращая в результате `bits` ❸. Наконец, при указании позиции (3) и длины (3) получается `bit` ❹.

Краткое описание методов работы со строками

В таблице 15.5 перечислены многие методы вставки и удаления строки. В этой таблице `str` — строка или `char*` в стиле C, `p` и `n` — `size_t`, `ind` — индекс `size_t` или итератор для `s`, `n` и `i` — `size_t`, `c` — `char`, и `beg` и `end` являются итераторами. Звездочка (*) указывает, что эта операция делает недействительными простые указатели и итераторы для элементов `s`, по крайней мере при некоторых обстоятельствах.

Таблица 15.5. Поддерживаемые методы работы с элементами `std::string`

Метод	Описание
`s.insert(ind, str, [p], [n])`	Вставляет `n` элементов `str`, начиная с `p`, в `s` непосредственно перед `ind`. Если `n` не указан, вставляет всю строку или до первого нуля `char*; p` по умолчанию равен 0*
`s.insert(ind, n, c)`	Вставляет `n` копий `c` непосредственно перед `ind` *
`s.insert(ind, beg, end)`	Вставляет полуоткрытый диапазон от `beg` до `end` перед `ind` *
`s.append(str, [p], [n])`	Эквивалент `s.insert (s.end(), str, [p], [n])` *
`s.append(n, c)`	Эквивалент `s.insert (s.end(), n, c)` *
`s.append(beg, end)`	Добавляет полуоткрытый диапазон от `beg` до `end` и до конца `s`
`s += c` `s += str`	Добавляет `c` или `str` в конец `s` *
`s.push_back(c)`	Добавляет `c` в конец `s` *
`s.clear()`	Удаляет все символы из `s` *
`s.erase([i], [n])`	Удаляет `n` символов, начиная с позиции `i`; по умолчанию `i` равен 0, и `n` по умолчанию — остаток от `s` *
`s.erase(itr)`	Стирает элемент, на который указывает `itr` *
`s.erase(beg, end)`	Стирает элементы в полуоткрытом диапазоне от `beg` до `end` *
`s.pop_back()`	Удаляет последний элемент `s` *
`s.resize(n, [c])`	Изменяет размер строки, чтобы она содержала `n` символов. Если эта операция увеличивает длину строки, добавляет копии `c`, которые по умолчанию равны 0 *
`s.replace(i, n1, str, [p], [n2])`	Заменяет `n1` символов, начиная с индекса `i`, с `n` 2 элементами в `str`, начиная с `p`. По умолчанию `p` равен 0, а `n` 2 равен `str.length()` *

Продолжение ⇩

Таблица 15.5 *(продолжение)*

Метод	Описание
`s.replace(beg, end, str)`	Заменяет полуоткрытый диапазон от `beg` до `end` на `str` *
`s.replace(p, n, str)`	Заменяет часть строки от `p` до `p + n` на `str` *
`s.replace(beg1, end1, beg2, end2)`	Заменяет полуоткрытый диапазон от `beg` 1 до `end` 1 полуоткрытым диапазоном от `beg` 2 до `end` 2 *
`s.replace(ind, c, [n])`	Заменяет `n` элементов, начиная с `ind`, на `cs` *
`s.replace(ind, beg, end)`	Заменяет элементы, начиная с `ind`, полуоткрытым диапазоном от `beg` до `end` *
`s.substr([p], [c])`	Возвращает подстроку, начинающуюся с `p` с длиной `c`. По умолчанию `p` равен 0, а `c` — остаток строки
`s1.swap(s2)` `swap(s1, s2)`	Меняет содержимое `s1` на `s2` *

Поиск

В дополнение к предыдущим методам `string` предлагает несколько *методов поиска*, которые позволяют найти нужные подстроки и символы. Каждый метод выполняет определенный вид поиска, поэтому его выбор зависит от особенностей приложения.

find

Первый метод, предлагаемый `string`, — это `find`, который принимает строку, строку в стиле C или `char` в качестве первого аргумента. Этот аргумент является элементом, который нужно найти в `this`. При желании можно указать второй аргумент `size_t`, который сообщает `find`, с чего начать поиск. Если команда `find` не может найти подстроку, она возвращает специальный постоянный статический `size_t` член `std::string::npos`. В листинге 15.13 показан метод `find`.

Листинг 15.13. Поиск подстрок в string

```
TEST_CASE("std::string find") {
  using namespace std::literals::string_literals;
  std::string word("pizzazz"); ❶
  SECTION("locates substrings from strings") {
    REQUIRE(word.find("zz"s) == 2); // pi(z)zazz ❷
  }
  SECTION("accepts a position argument") {
    REQUIRE(word.find("zz"s, 3) == 5); // pizza(z)z ❸
  }
  SECTION("locates substrings from char*") {
    REQUIRE(word.find("zaz") == 3); // piz(z)azz ❹
  }
  SECTION("returns npos when not found") {
    REQUIRE(word.find('x') == std::string::npos); ❺
  }
}
```

Здесь создается строка под названием `word`, содержащая `pizzazz` ❶. В первом тесте вызывается `find` со строкой, содержащей `zz`, которая возвращает 2 ❷, индекс первого `z` в `pizzazz`. Когда предоставляется аргумент позиции 3, соответствующий второму `z` в `pizzazz`, `find` находит второй `zz`, начинающийся с 5 ❸. В третьем тесте используется строка `zaz` в стиле C и находятся возвращаемые значения 3, снова соответствующие второму `z` в `pizzazz` ❹. Наконец, программа пытается найти символ `x`, которого нет в `pizzazz`, поэтому `find` возвращает `std::string::npos` ❺.

rfind

Метод `rfind` является альтернативой `find`, которая принимает те же аргументы, но выполняет поиск *в обратном порядке*. Возможно, вы захотите использовать эту функцию, например для нахождения конкретного знака препинания в конце `string`, как показывает листинг 15.14.

Листинг 15.14. Поиск подстрок в строке в обратном порядке

```
TEST_CASE("std::string rfind") {
  using namespace std::literals::string_literals;
  std::string word("pizzazz"); ❶
  SECTION("locates substrings from strings") {
    REQUIRE(word.rfind("zz"s) == 5); // pizza(z)z ❷
  }
  SECTION("accepts a position argument") {
    REQUIRE(word.rfind("zz"s, 3) == 2); // pi(z)zazz ❸
  }
  SECTION("locates substrings from char*") {
    REQUIRE(word.rfind("zaz") == 3); // piz(z)azz ❹
  }
  SECTION("returns npos when not found") {
    REQUIRE(word.rfind('x') == std::string::npos); ❺
  }
}
```

С тем же `word` ❶ используются те же аргументы, что и в листинге 15.13, для проверки `rfind`. Учитывая `zz`, `rfind` возвращает 5, второй за последним `z` в `pizzazz` ❷. Когда предоставляется аргумент позиции 3, вместо этого `rfind` возвращает первый `z` в `pizzazz` ❸. Поскольку есть только одно вхождение подстроки `zaz`, `rfind` возвращает ту же позицию, что и `find` ❹. Так же как и `find`, `rfind` возвращает `std::string::npos`, когда задано `x` ❺.

find_*_of

В то время как `find` и `rfind` определяют местонахождение точных подстрок в `string`, семейство связанных функций находит первый символ, содержащийся в данном аргументе. Функция `find_first_of` принимает `string` и находит первый символ, содержащийся в аргументе. При желании можно указать аргумент `size_t`, чтобы указать `find_first_of`, где начинать строку. Если `find_first_of` не может найти соответствующий символ, то вернет `std::string::npos`. В листинге 15.15 показана функция `find_first_of`.

Листинг 15.15. Поиск первого элемента из набора в строке

```
TEST_CASE("std::string find_first_of") {
  using namespace std::literals::string_literals;
  std::string sentence("I am a Zizzer-Zazzer-Zuzz as you can plainly see."); ❶
  SECTION("locates characters within another string") {
    REQUIRE(sentence.find_first_of("Zz"s) == 7); // (Z)izzer❷
  }
  SECTION("accepts a position argument") {
    REQUIRE(sentence.find_first_of("Zz"s, 11) == 14); // (Z)azzer ❸
  }
  SECTION("returns npos when not found") {
    REQUIRE(sentence.find_first_of("Xx"s) == std::string::npos); ❹
  }
}
```

Строка `sentence` содержит `I am a Zizzer-Zazzer-Zuzz as you can plainly see.` ❶. Здесь вызывается `find_first_of` со строкой `Zz`, которая соответствует строчной и прописной букве `z`. Этот вызов возвращает 7, что соответствует первому `Z` в `sentence`, `Zizzer` ❷. Во втором тесте снова предоставляется строка `Zz`, но также передается аргумент позиции 11, который соответствует `e` в `Zizzer`. Это приводит к 14, что соответствует `Z` в `Zazzer` ❸. Наконец вызывается `find_first_of` с аргументом `Xx`, что приводит к `std::string::npos`, потому что `sentence` не содержит `x` (или `X`) ❹.

`string` предлагает три варианта `find_first_of`:

- `find_first_not_of` возвращает первый символ, не содержащийся в строковом аргументе. Вместо предоставления строки, содержащей элементы, которые нужно найти, предоставляется `string` символов, которые не нужно находить;
- `find_last_of` выполняет сопоставление в обратном порядке; вместо поиска в начале `string` или с аргумента позиции и перехода к концу, `find_last_of` начинается с конца `string` или аргумента позиции и продолжает выполнение до начала;
- `find_last_not_of` объединяет два предыдущих варианта: передается строка, содержащая элементы, которые не нужно находить, и `find_last_not_of` выполняет поиск в обратном порядке.

Ваш выбор функции `find` сводится к тому, каковы алгоритмические требования. Вам нужно искать с конца строки, например по знаку препинания? Если да, используйте `find_last_of`. Нужно найти первое появление в строке? Тогда используйте `find_first_of`. Нужно инвертировать поиск и найти первый элемент, который не входит в какой-либо набор? Используйте альтернативы `find_first_not_of` и `find_last_not_of` в зависимости от того, нужно ли начать с начала или с конца строки.

В листинге 15.16 приведены эти три варианта `find_first_of`.

Листинг 15.16. Альтернативы методу find_first_of в string

```
TEST_CASE("std::string") {
  using namespace std::literals::string_literals;
  std::string sentence("I am a Zizzer-Zazzer-Zuzz as you can plainly see."); ❶
```

```
  SECTION("find_last_of finds last element within another string") {
    REQUIRE(sentence.find_last_of("Zz"s) == 24); // Zuz(z) ❷
  }
  SECTION("find_first_not_of finds first element not within another string") {
    REQUIRE(sentence.find_first_not_of(" -IZaeimrz"s) == 22); // Z(u)zz ❸
  }
  SECTION("find_last_not_of finds last element not within another string") {
    REQUIRE(sentence.find_last_not_of(" .es"s) == 43); // plainl(y) ❹
  }
}
```

Здесь инициализируется тот же `sentence`, что и в листинге 15.15 ❶. В первом тесте используется `find_last_of` для `Zz`, который ищет в обратном порядке любые `z` или `Z` и возвращает 24, последнее `z` в `Zuzz` ❷. Далее используется `find_first_not_of` и передается множество символов (не включая букву `u`), что приводит к 22 — положению первого символа `u` в `Zuzz` ❸. Наконец, используется `find_last_not_of`, чтобы найти последний символ, не равный пробелу, точке, `e` или `s`. Это приводит к 43 — положение `y` в `plainly` ❹.

Краткое изложение методов поиска

В таблице 15.6 перечислены многие методы поиска в `string`. Обратите внимание, что `s2` означает строку; `cstr` — строка `char*` в стиле C; `c` — символ; а `n`, `l` и `p` — это `size_t` в таблице.

Таблица 15.6. Поддерживаемые алгоритмы поиска `std::string`

Метод	Ищет s, начиная с p, и возвращает позицию…
`s.find(s2, [p])`	Первой подстроки, равной `s2`; `p` по умолчанию равен 0
`s.find(cstr, [p], [l])`	Первой подстроки, равной первым `l` символам `cstr`; `p` по умолчанию 0; `l` по умолчанию равен длине `cstr` до нулевого символа конца строки
`s.find(c, [p])`	Первого символа, равного `c`; `p` по умолчанию 0
`s.rfind(s2, [p])`	Последней подстроки, равной `s2`; `p` по умолчанию равен `npos`
`s.rfind(cstr, [p], [l])`	Последней подстроки, равной первым `l` символам `cstr`; `p` по умолчанию равен `npos`; `l` по умолчанию равен длине `cstr` до нулевого символа конца строки
`s.rfind(c, [p])`	Последнего символа, равного `c`; `p` по умолчанию равен `npos`
`s.find_first_of(s2, [p])`	Первого символа, содержащегося в `s2`; `p` по умолчанию равен 0
`s.find_first_of(cstr, [p], [l])`	Первого символа, содержащегося в первых `l` символах `cstr`; `p` по умолчанию равен 0; `l` по умолчанию равен длине `cstr` до нулевого символа конца строки
`s.find_first_of(c, [p])`	Первого символа, равного `c`; `p` по умолчанию равен 0

Продолжение ⇩

Таблица 15.6 *(продолжение)*

Метод	Ищет s, начиная с p, и возвращает позицию...
`s.find_last_of(s2, [p])`	Последнего символа, содержащегося в s2; p по умолчанию равен 0
`s.find_last_of(cstr, [p], [l])`	Последнего символа, содержащегося в первых l символах cstr; p по умолчанию равен 0; l по умолчанию равен длине cstr до нулевого символа конца строки
`s.find_last_of(c, [p])`	Последнего символа, равного c; p по умолчанию равен 0
`s.find_first_not_of(s2, [p])`	Первого символа, не содержащегося в s2; p по умолчанию равен 0
`s.find_first_not_of (cstr, [p], [l])`	Первого символа, не содержащегося в первых l символах cstr; p по умолчанию равен 0; l по умолчанию равен длине cstr до нулевого символа конца строки
`s.find_first_not_of(c, [p])`	Первого символа, не равного c; p по умолчанию равен 0
`s.find_last_not_of(s2, [p])`	Последнего символа, не содержащегося в s2; p по умолчанию равен 0
`s.find_last_not_of (cstr, [p], [l])`	Последнего символа, не содержащегося в первых l символах cstr; p по умолчанию равен 0; l по умолчанию равен длине cstr до нулевого символа конца строки
`s.find_last_not_of(c, [p])`	Последнего символа, не равного c; p по умолчанию равен 0

Числовые преобразования

STL предоставляет функции для преобразования между `string` или `wstring` и основными числовыми типами. Учитывая числовой тип, можно использовать функции `std::to_string` и `std::to_wstring` для генерации его `string` или `wstring` представления. Обе функции имеют перегрузки для всех числовых типов.

В листинге 15.17 показаны `string` и `wstring`.

Листинг 15.17. Числовые функции преобразования string

```
TEST_CASE("STL string conversion function") {
  using namespace std::literals::string_literals;
  SECTION("to_string") {
    REQUIRE("8675309"s == std::to_string(8675309)); ❶
  }
  SECTION("to_wstring") {
    REQUIRE(L"109951.1627776"s == std::to_wstring(109951.1627776)); ❷
  }
}
```

ПРИМЕЧАНИЕ

Из-за неточности типа `double` второй юнит-тест ❷ может не сработать в вашей системе.

Первый пример использует `to_string` для преобразования `int8675309` в `string` ❶; второй пример использует `to_wstring` для преобразования `double109951.1627776` в `wstring` ❷.

Также можно преобразовать другим способом, перейдя от `string` или `wstring` к числовому типу. Каждая функция преобразования чисел принимает в качестве первого аргумента `string` или `wstring`, содержащую число в формате строки. Далее можно предоставить необязательный указатель `size_t`. Если это предусмотрено, функция преобразования запишет индекс последнего символа, который она смогла преобразовать (или длину входной строки, если она декодировала все символы). По умолчанию этот аргумент индекса равен `nullptr`, и в этом случае функция преобразования не записывает индекс. Когда целевой тип является целым, можно предоставить третий аргумент: `int`, соответствующий основанию закодированной строки. Этот базовый аргумент является необязательным и по умолчанию равен 10.

Каждая функция преобразования выдает `std::invalid_argument`, если преобразование не может быть выполнено, и `std::out_of_range`, если преобразованное значение выходит за пределы диапазона для соответствующего типа.

Таблица 15.7 перечисляет каждую из этих функций преобразования вместе с ее целевым типом. В этой таблице `s` означает строку. Если `p` не равен `nullptr`, функция преобразования запишет позицию первого непреобразованного символа в `s` в память, на которую указывает `p`. Если все символы закодированы, возвращается длина `s`. Здесь `b` — базовое представление числа в `s`. Обратите внимание, что по умолчанию `p` равен `nullptr`, а `b` — 10.

Таблица 15.7. Поддерживаемые функции преобразования чисел для `std::string` и `std::wstring`

Функция	Преобразовывает `s` в ...
`stoi(s, [p], [b])`	`int`
`stol(s, [p], [b])`	`long`
`stoll(s, [p], [b])`	`long long`
`stoul(s, [p], [b])`	`unsigned long`
`stoull(s, [p], [b])`	`unsigned long long`
`stof(s, [p])`	`float`
`stod(s, [p])`	`double`
`stold(s, [p])`	`long double`
`to_string(n)`	`string`
`to_wstring(n)`	`wstring`

Листинг 15.18 показывает несколько числовых функций преобразования.

Листинг 15.18. Функции преобразования string

```
TEST_CASE("STL string conversion function") {
  using namespace std::literals::string_literals;
  SECTION("stoi") {
    REQUIRE(std::stoi("8675309"s) == 8675309); ❶
  }
  SECTION("stoi") {
    REQUIRE_THROWS_AS(std::stoi("1099511627776"s), std::out_of_range); ❷
  }
  SECTION("stoul with all valid characters") {
    size_t last_character{};
    const auto result = std::stoul("0xD3C34C3D"s, &last_character, 16); ❸
    REQUIRE(result == 0xD3C34C3D);
    REQUIRE(last_character == 10);
  }
  SECTION("stoul") {
    size_t last_character{};
    const auto result = std::stoul("42six"s, &last_character); ❹
    REQUIRE(result == 42);
    REQUIRE(last_character == 2);
  }
  SECTION("stod") {
    REQUIRE(std::stod("2.7182818"s) == Approx(2.7182818)); ❺
  }
}
```

Во-первых, `stoi` используется для преобразования `8675309` в целое число ❶. Во втором тесте происходит попытка использовать `stoi` для преобразования `string 1099511627776` в целое число. Поскольку это значение слишком велико для `int`, `stoi` выдает `std::out_of_range` ❷. Затем `0xD3C34C3D` конвертируется с помощью `stoi`, но предоставляются два необязательных аргумента: указатель на `size_t` с именем `last_character` и шестнадцатеричное основание ❸. Объект `last_character` равен 10, длина — `0xD3C34C3D`, потому что `stoi` может анализировать каждый символ. `string` в следующем тесте, `42six`, содержит не поддающиеся синтаксическому анализу символы `six`. При вызове `stoul` на этот раз результат равен 42, а `last_character` равно 2, позиция `s` в `six` ❹. Наконец, используется `stod` для преобразования `string2.7182818` в `double` ❺.

ПРИМЕЧАНИЕ

Boost Lexical Cast предоставляет альтернативный, основанный на шаблонах подход к числовым преобразованиям. Обратитесь к документации для `boost::lexical_cast`, доступного в заголовке `<boost/lexical_cast.hpp>`.

Строковое представление

Строковое представление — это объект, представляющий постоянную непрерывную последовательность символов. Это очень похоже на `const string` ссылку.

Фактически классы строкового представления часто реализуются как указатель на последовательность символов и длину.

STL предлагает шаблон класса `std::basic_string_view` в заголовке `<string _view>`, который аналогичен `std::basic_string`. Шаблон `std::basic_string_view` имеет специализацию для каждого из четырех обычно используемых типов символов:

- `char` имеет `string_view`;
- `wchar_t` имеет `wstring_view`;
- `char16_t` имеет `u16string_view`;
- `char32_t` имеет `u32string_view`.

В этом разделе в целях демонстрации обсуждается специализация `string_view`, однако это обсуждение обобщается на остальные три специализации.

Класс `string_view` поддерживает большинство тех же методов, что и `string`; на самом деле он предназначен для замены `const string&`.

Создание

Класс `string_view` поддерживает конструктор по умолчанию, поэтому он имеет нулевую длину и указывает на `nullptr`. Важно отметить, что `string_view` поддерживает неявный конструктор из `const string&` или строки в стиле C. Можно создать `string_view` из `char*` и `size_t`, так что можно вручную указать желаемую длину, если нужна подстрока или есть встроенные нули. В листинге 15.19 показано использование `string_view`.

Листинг 15.19. Конструкторы string_view

```
TEST_CASE("std::string_view supports") {
  SECTION("default construction") {
    std::string_view view; ❶
    REQUIRE(view.data() == nullptr);
    REQUIRE(view.size() == 0);
    REQUIRE(view.empty());
  }
  SECTION("construction from string") {
    std::string word("sacrosanct");
    std::string_view view(word); ❷
    REQUIRE(view == "sacrosanct");
  }
  SECTION("construction from C-string") {
    auto word = "viewership";
    std::string_view view(word); ❸
    REQUIRE(view == "viewership");
  }
  SECTION("construction from C-string and length") {
    auto word = "viewership";
    std::string_view view(word, 4); ❹
    REQUIRE(view == "view");
  }
}
```

Построенный по умолчанию `string_view` указывает на `nullptr` и является пустым ❶. При создании представления `string_view` из `string` ❷ или строки в стиле C ❸ оно указывает на содержимое оригинала. Последний тест предоставляет необязательный аргумент длины 4, что означает, что `string_view` ссылается только на первые четыре символа вместо этого ❹.

Несмотря на то что `string_view` также поддерживает конструктор и присваивание копирования, он не поддерживает конструктор или присваивание перемещения. Такой дизайн имеет смысл, если иметь в виду, что `string_view` не принадлежит последовательность, на которую указывает.

Поддерживаемые операции string_view

Класс `string_view` поддерживает многие из тех же операций, что и `const string`, и с идентичной семантикой. Ниже перечислены все общие методы между `string` и `string_view`:

- **Итераторы**

  `begin`, `end`, `rbegin`, `rend`, `cbegin`, `cend`, `crbegin`, `crend`

- **Доступ к элементам**

  `operator[]`, `at`, `front`, `back`, `data`

- **Емкость**

  `size`, `length`, `max_size`, `empty`

- **Поиск**

  `find`, `rfind`, `find_first_of`, `find_last_of`, `find_first_not_of`, `find_last_not_of`

- **Извлечение**

  `copy`, `substr`

- **Сравнение**

  `compare`, `operator==`, `operator!=`, `operator<`, `operator>`, `operator<=`, `operator>=`

В дополнение к этим общим методам `string_view` поддерживает метод `remove_prefix`, который удаляет указанное количество символов из начала `string_view`, и метод `remove_suffix`, который вместо этого удаляет символы с конца. В листинге 15.20 приведены оба метода.

Листинг 15.20. Изменение string_view с помощью remove_prefix и remove_suffix

```
TEST_CASE("std::string_view is modifiable with") {
  std::string_view view("previewing"); ❶
  SECTION("remove_prefix") {
    view.remove_prefix(3); ❷
    REQUIRE(view == "viewing");
  }
```

```
  SECTION("remove_suffix") {
    view.remove_suffix(3); ❸
    REQUIRE(view == "preview");
  }
}
```

Здесь объявляется `string_view` со ссылкой на строковый литерал `previewing` ❶. Первый тест вызывает `remove_prefix` с аргументом 3 ❷, который удаляет три символа с начала `string_view`, поэтому теперь тот имеет значение `viewing`. Второй тест вместо этого вызывает `remove_suffix` с 3 ❸, который удаляет три символа из задней части `string_view` и приводит к результату `preview`.

Владение, использование и эффективность

Поскольку `string_view` не принадлежит последовательность, к которой он относится, нужно убедиться, что время жизни `string_view` является подмножеством времени жизни упомянутой последовательности.

Возможно, наиболее распространенное использование `string_view` — это параметр функции. Когда нужно взаимодействовать с неизменной последовательностью символов, `string_view` можно использовать в качестве первого прибежища. Рассмотрим функцию `count_vees` в листинге 15.21, которая подсчитывает частоту буквы `v` в последовательности символов.

Листинг 15.21. Функция count_vees

```
#include <string_view>

size_t count_vees(std::string_view my_view❶) {
  size_t result{};
  for(auto letter : my_view) ❷
    if (letter == 'v') result++; ❸
  return result; ❹
}
```

Функция `count_vees` принимает `string_view` с именем `my_view` ❶, который повторяется, используя цикл `for` ❷ на основе диапазона. Каждый раз, когда символ в `my_view` равен `v`, результирующая переменная ❸ увеличивается и возвращается после исчерпания последовательности ❹.

Можно переопределить листинг 15.21, просто заменив `string_view` на `const string&`, как показано в листинге 15.22.

Листинг 15.22. Функция count_vees переопределена для использования константной const string& вместо string_view

```
#include <string>

size_t count_vees(const std::string& my_view) {
--пропуск--
}
```

Если `string_view` является просто заменой `const string&`, зачем ее использовать? Что ж, если вы вызываете `count_vees` с помощью `std::string`, нет никакой разницы: современные компиляторы будут генерировать тот же код.

Если вместо этого вызывается `count_vees` со строковым литералом, есть большая разница: при передаче строкового литерала в `const string&` создается `string`. При передаче строкового литерала в `string_view` создается `string_view`. Построение строки, вероятно, более затратно, поскольку может потребоваться выделение динамической памяти и, безусловно, копирование символов. `string_view` — это просто указатель и длина (копирование или выделение памяти не требуется).

Регулярные выражения

Регулярное выражение, или *regex*, является строкой, которая определяет шаблон поиска. Регулярные выражения имеют долгую историю в информатике и образуют своего рода мини-язык для поиска, замены и извлечения языковых данных. STL предлагает поддержку регулярных выражений в заголовке `<regex>`.

При правильном использовании регулярные выражения могут быть чрезвычайно мощными, декларативными и краткими; Тем не менее легко написать регулярные выражения, которые являются абсолютно непостижимыми. Используйте регулярные выражения с умом.

Шаблоны

Регулярные выражения создаются при помощи строк, называемых *шаблонами*. Шаблоны представляют желаемый набор строк с использованием определенной грамматики регулярных выражений, которая устанавливает синтаксис для построения шаблонов. Другими словами, шаблон определяет подмножество всех возможных строк, которые вас интересуют. STL поддерживает несколько грамматик, но здесь основное внимание будет уделено основам грамматики по умолчанию, модифицированной грамматике регулярных выражений ECMAScript (см. [re.grammar] для деталей).

Классы символов

В грамматике ECMAScript смешиваются буквенные символы со специальной разметкой для описания желаемых строк. Возможно, наиболее распространенной разметкой является *класс символов*, который заменяет собой набор возможных символов: `\d` соответствует любой цифре, `\s` соответствует любому пробельному символу, а `\w` соответствует любому буквенно-цифровому («словесному») символу.

В таблице 15.8 приведено несколько примеров регулярных выражений и возможных интерпретаций.

Таблица 15.8. Шаблоны регулярных выражений, использующие только классы символов и литералы

Шаблон регулярного выражения	Может описывать
`\d\d\d-\d\d\d-\d\d\d\d`	Американский формат номера телефона, например `202-456-1414`
`\d\d:\d\d \wM`	Время в формате `ЧЧ:MM AM/PM`, например `08:49 PM`
`\w\w\d\d\d\d\d\d`	Американский формат почтового индекса, включающий код штата, например `NJ07932`
`\w\d-\w\d`	Идентификатор астрономического дроида, например `R2-D2`
`c\wt`	Трехбуквенное слово, начинающееся с `c` и заканчивающееся `t`, например `cat` или `cot`

Также можно инвертировать класс символов, используя заглавные буквы *d*, *s* или *w*, чтобы получить обратное: `\D` соответствует любому нецифровому символу, `\S` соответствует любому не пробельному символу, а `\W` соответствует любому несловесному символу.

Кроме того, можно создавать свои собственные классы символов, явно перечисляя их в квадратных скобках `[]`. Например, класс символов `[02468]` включает четные цифры. Также можно использовать дефисы в качестве ярлыков для включения подразумеваемых диапазонов, поэтому класс символов `[0-9a-fA-F]` включает любую шестнадцатеричную цифру независимо от того, является буква заглавной или нет. Наконец, можно инвертировать пользовательский класс символов, добавив в список знак вставки `^`. Например, класс символов `[^aeiou]` включает в себя все не гласные символы.

Квантификаторы

Можно избавиться от печатания некоторых символов с помощью *квантификаторов*, которые указывают, что символ слева должен повторяться несколько раз. В таблице 15.9 перечислены квантификаторы регулярных выражений.

Таблица 15.9. Квантификаторы регулярных выражений

Квантификатор регулярного выражения	Определяет количество
`*`	0 или больше
`+`	1 или больше
`?`	0 или 1
`{n}`	Ровно `n`
`{n, m}`	Между `n` и `m` включительно
`{n, }`	Как минимум `n`

Используя квантификаторы, можно указать все слова, начинающиеся с *c* и заканчивающиеся на *t*, используя шаблон `c\w*t`, потому что `\w*` соответствует любому количеству символов слова.

Группы

Группа — это коллекция символов. Можно указать группу, поместив ее в скобки. Группы полезны в нескольких случаях, включая указание конкретной коллекции для возможного извлечения и для количественного определения.

Например, можно улучшить шаблон ZIP в таблице 15.8, чтобы использовать квантификаторы и группы, например:

```
(\w{2})?❶(\d{5})❷(-\d{4})?❸
```

Теперь у вас есть три группы: необязательный код штата ❶, почтовый индекс ❷ и необязательный четырехзначный суффикс ❸. Как вы увидите позже, эти группы значительно упрощают синтаксический анализ регулярных выражений.

Другие специальные символы

В таблице 15.10 перечислены несколько других специальных символов, доступных для использования в шаблонах регулярных выражений.

Таблица 15.10. Примеры специальных символов

Символ	Определяет
`X|Y`	Символ `X` или `Y`
`\Y`	Специальный символ `Y` как литерал (другими словами, экранирование)
`\n`	Новую строку
`\r`	Возврат каретки
`\t`	Символ табуляции
`\0`	Ноль
`\xYY`	Шестнадцатеричный символ, соответствующий `YY`

basic_regex

Шаблон класса в STL `std::basic_regex` в заголовке `<regex>` представляет регулярное выражение, построенное из шаблона. Класс `basic_regex` принимает два параметра шаблона, тип символа и необязательный класс типажей. Вы почти всегда будете использовать одну из вспомогательных специализаций: `std::regex` для `std::basic_regex<char>` или `std::wregex` для `std::basic_regex<wchar_t>`.

Основным средством построения регулярного выражения является передача строкового литерала, содержащего шаблон регулярного выражения. Поскольку для

шаблонов потребуется много экранированных символов, особенно обратной косой черты \, рекомендуется использовать обычные строковые литералы, например `R"()"`. Конструктор принимает второй необязательный параметр для указания синтаксических флагов, таких как грамматика регулярных выражений.

Хотя `regex` используется главным образом в качестве входных данных в алгоритмах регулярных выражений, оно предлагает несколько методов, с которыми пользователи могут взаимодействовать. Оно поддерживает обычные конструкторы и присваивания копирования и переноса, `swap`, а также следующее:

- `assign(s)` переназначает шаблон на `s`;
- `mark_count()`возвращает количество групп в шаблоне;
- `flags()`возвращает синтаксические флаги, выданные в конструкторе.

В листинге 15.23 показано, как можно создать регулярное выражение для почтового индекса и проверить его подгруппы.

Листинг 15.23. Построение регулярного выражения с использованием обычного строкового литерала и извлечение его счетчика групп

```
#include <regex>

TEST_CASE("std::basic_regex constructs from a string literal") {
  std::regex zip_regex{ R"((\w{2})?(\d{5})(-\d{4})?)" }; ❶
  REQUIRE(zip_regex.mark_count() == 3); ❷
}
```

Здесь создается регулярное выражение с именем `zip_regex` с использованием `(\w{2})?(\d{5})(-\d{4})?` ❶. Используя метод `mark_count`, вы видите, что `zip_regex` содержит три группы ❷.

Алгоритмы

Класс `<regex>` содержит три алгоритма для применения `std::basic_regex` к целевой строке: сопоставление, поиск и замену. То, что вы выберете, зависит от поставленной задачи.

Сопоставление

Сопоставление пытается соединить регулярное выражение со *всей* строкой. STL предоставляет функцию `std::regex_match` для сопоставления, которая имеет четыре перегрузки.

Во-первых, можно предоставить в `regex_match string`, C-строку или начальный и конечный итератор, образующий полуоткрытый диапазон. Следующий параметр — это необязательный атрибут объекта `std::match_results`, который получает сведения о совпадении. Следующий параметр — это `std::basic_regex`, который определяет сопоставление, а последний параметр — необязательный `std::regex_constants::match_flag_type`, который указывает дополнительные параметры сопо-

ставления для расширенных вариантов использования. Функция `regex_match` возвращает значение типа `bool`, которое имеет значение `true`, если найдено совпадение; в противном случае это — `false`.

Подводя итог, можно вызвать `regex_match` следующими способами:

```
regex_match(beg, end, [mr], rgx, [flg])
regex_match(str, [mr], rgx, [flg])
```

Либо укажите полуоткрытый диапазон от `beg` до `end`, либо `string`/C-строку `str` для поиска. При желании можно предоставить `match_results` с именем `mr` для хранения всех деталей любых найденных совпадений. Очевидно, нужно предоставить регулярное выражение `rgx`. Наконец, флаги `flg` используются редко.

ПРИМЕЧАНИЕ

Подробнее о флагах сопоставлений `flg` см. в [re.alg.match].

Подсовпадение — это подстрока строки сопоставления, которая соответствует группе. Почтовый индекс, соответствующий регулярному выражению `(\w{2})(\d{5})(-\d{4})?`, может создать два или три подсовпадения в зависимости от строки. Например, TX78209 содержит два подсовпадения: TX и 78209, а NJ07936-3173 содержит три подсовпадения: NJ, 07936 и -3173.

Класс `match_results` хранит ноль или более экземпляров `std::sub_match`. `sub_match` — это простой шаблон класса, который предоставляет метод `length`, чтобы возвратить длину подсовпадения, и метод `str`, чтобы построить `string` из `sub_match`.

Несколько сбивает с толку, если `regex_match` успешно совпадает со строкой, `match_results` сохраняет всю совпавшую строку как свой первый элемент, а затем сохраняет любые подсовпадения как последующие элементы.

Класс `match_results` предоставляет операции, перечисленные в таблице 15.11.

Таблица 15.11. Поддерживаемые операции `match_results`

Операция	Описание
`mr.empty()`	Проверяет, было ли сопоставление успешным
`mr.size()`	Возвращает количество подсовпадений
`mr.max_size()`	Возвращает максимальное количество подсовпадений
`mr.length([i])`	Возвращает длину подсовпадения `i`, которая по умолчанию равна 0
`mr.position([i])`	Возвращает символ первой позиции подсовпадения `i`, который по умолчанию равен 0
`mr.str([i])`	Возвращает строку, представляющую подсовпадение `i`, которая по умолчанию равна 0

Операция	Описание
`mr[i]`	Возвращает ссылку на класс `std::sub_match`, соответствующий подсовпадению `i`, который по умолчанию равен 0
`mr.prefix()`	Возвращает ссылку на класс `std::sub_match`, соответствующий последовательности перед подсовпадением
`mr.suffix()`	Возвращает ссылку на класс `std::sub_match`, соответствующий последовательности после подсовпадения
`mr.format(str)`	Возвращает строку с содержимым в соответствии с форматной строкой `str`. Существует три специальных последовательности: `$`` для символов перед совпадением, `$´` для символов после совпадения и `$&` для символов совпадения

`mr.begin()`
`mr.end()`
`mr.cbegin()`
`mr.cend()`
	Возвращает соответствующий итератор в последовательности совпадений

Шаблон класса `std::sub_match` имеет предопределенные специализации для работы с общими типами строк:

- `std::csub_match` для `const char*`;
- `std::wcsub_match` для `const wchar_t*`;
- `std::ssub_match` для `std::string`;
- `std::wssub_match` для `std::wstring`.

К сожалению, придется отслеживать все эти специализации вручную из-за дизайна `std::regex_match`. Этот дизайн, как правило, смущает новичков, поэтому давайте рассмотрим пример. Листинг 15.24 использует регулярное выражение почтового индекса `(\w{2})(\d{5})(-\d{4})?`, чтобы проверять соответствие строк `NJ07936-3173` и `Iomega Zip 100`.

Листинг 15.24. regex_match пытается сопоставить регулярное выражение со строкой

```
#include <regex>
#include <string>

TEST_CASE("std::sub_match") {
  std::regex regex{ R"((\w{2})(\d{5})(-\d{4})?)" }; ❶
  std::smatch results; ❷
  SECTION("returns true given matching string") {
    std::string zip("NJ07936-3173");
    const auto matched = std::regex_match(zip, results, regex); ❸
    REQUIRE(matched); ❹
    REQUIRE(results[0] == "NJ07936-3173"); ❺
    REQUIRE(results[1] == "NJ"); ❻
    REQUIRE(results[2] == "07936");
```

```
    REQUIRE(results[3] == "-3173");
  }
  SECTION("returns false given non-matching string") {
    std::string zip("Iomega Zip 100");
    const auto matched = std::regex_match(zip, results, regex); ❼
    REQUIRE_FALSE(matched); ❽
  }
}
```

Создается регулярное выражение с обычным литералом `R"((\w{2})(\d{5})(-\d{4})?)"` ❶, и по умолчанию создается `smatch` ❷. В первом тесте вызывается `regex_match` для корректного почтового индекса `NJ07936-3173` ❸, который возвращает `matched` со значением `true`, чтобы указать на успешное совпадение ❹. Поскольку `smatch` предоставляется для `regex_match`, он содержит корректный почтовый индекс в качестве первого элемента ❺, за которым следует каждая из трех подгрупп ❻.

Во втором тесте используется `regex_match` для неверного почтового индекса `Iomega Zip 100` ❼, который не выдает совпадения и возвращает `false` ❽.

Поиск

Поиск пытается сопоставить регулярное выражение с *частью* строки. STL предоставляет функцию поиска `std::regex_search`, которая по сути является заменой для `regex_match`, которая успешно выполняется, даже когда регулярному выражению соответствует только часть строки.

Например, строка `The string NJ07936-3173 is a ZIP Code` содержит почтовый индекс, но применение к нему регулярного выражения ZIP с использованием `std::regex_match` вернет `false`, поскольку регулярное выражение не соответствует *всей* строке. Тем не менее применение `std::regex_search` приведет к значению `true`, так как строка содержит корректный почтовый индекс. В листинге 15.25 показаны `regex_match` и `regex_search`.

Как и прежде, создается ZIP `regex` ❶. Также создается пример строки `sentence`, в которую вставляется корректный почтовый индекс ❷. Первый тест вызывает `regex_match` с `sentence` и `regex`, который возвращает `false` ❸. Второй тест вместо этого вызывает `regex_search` с теми же аргументами и возвращает `true` ❹.

Листинг 15.25. Сравнение regex_match и regex_search

```
TEST_CASE("when only part of a string matches a regex, std::regex_ ") {
  std::regex regex{ R"((\w{2})(\d{5})(-\d{4})?)" }; ❶
  std::string sentence("The string NJ07936-3173 is a ZIP Code."); ❷
  SECTION("match returns false") {
    REQUIRE_FALSE(std::regex_match(sentence, regex)); ❸
  }
  SECTION("search returns true") {
    REQUIRE(std::regex_search(sentence, regex)); ❹
  }
}
```

Замена

Замена заменяет вхождения регулярного выражения текстом для замены. STL предоставляет функцию `std::regex_replace` для замены.

В самом основном использовании `regex_replace` передаются три аргумента:

- строка/C-строка/полуоткрытый диапазон источника для поиска;
- регулярное выражение;
- строка для замены.

Например, в листинге 15.26 все гласные во фразе `queueing and cooeeing in eutopia` заменяются подчеркиванием (`_`).

Листинг 15.26. Использование std::regex_replace для замены подчеркиваний гласными в строке

```
TEST_CASE("std::regex_replace") {
  std::regex regex{ "[aeoiu]" }; ❶
  std::string phrase("queueing and cooeeing in eutopia"); ❷
  const auto result = std::regex_replace(phrase, regex, "_"); ❸
  REQUIRE(result == "q_____ng _nd c_____ng _n __t_p__"); ❹
}
```

Создается `std::regex`, который содержит набор всех гласных ❶, и `string` с именем `phrase`, содержащую контент `queueing and cooeeing in eutopia` ❷, богатый гласными. Затем вызывается `std::regex_replace` с `phrase` регулярным выражением и строковым литералом `_` ❸, который заменяет все гласные на подчеркивания ❹.

ПРИМЕЧАНИЕ

Boost Regex обеспечивает поддержку регулярных выражений, отражающих STL в заголовке `<boost/regex.hpp>`. Другая библиотека Boost, Xpressive, предлагает альтернативный подход с регулярными выражениями, которые можно выразить непосредственно в коде C++. У него есть некоторые основные преимущества, такие как проверка синтаксиса во время компиляции, но синтаксис обязательно отличается от стандартных синтаксисов регулярных выражений, таких как POSIX, Perl иECMAScript.

Библиотека Boost String Algorithms

Библиотека Boost String Algorithms (строковые алгоритмы Boost) предлагает множество функций для работы со `string`. Она содержит функции для общих задач, связанных со `string`, таких как обрезка, преобразование регистра, поиск/замена и вычисление характеристик. Доступ ко всем функциям Boost String Algorithms можно получить в пространстве имен `boost::algorithm` и во вспомогательном заголовке `<boost/attribute/string.hpp>`.

Boost Range

Range (диапазон) — это концепт (в смысле полиморфизма во время компиляции в главе 6), который имеет начало и конец, что позволяет перебирать составляющие элементы. Целью диапазона является улучшение практики прохождения полуоткрытого диапазона в виде пары итераторов. Заменив пару одним объектом, можно *составлять* алгоритмы вместе, используя результат диапазона одного алгоритма в качестве входных данных для другого. Например, если нужно преобразовать диапазон строк в верхний регистр и отсортировать их, можно передать результаты одной операции непосредственно в другую. Обычно это невозможно сделать только с помощью итераторов.

Диапазоны в настоящее время не являются частью стандарта C++, но существует несколько экспериментальных реализаций. Одной из таких реализаций является Boost Range, и поскольку строковые алгоритмы Boost широко используют Boost Range, давайте посмотрим на него сейчас.

Концепция Boost Range похожа на концепцию контейнера STL. Он обеспечивает обычное дополнение методов `begin/end` для представления итераторов по элементам в диапазоне. Каждый диапазон имеет категорию обхода, которая указывает поддерживаемые операции диапазона:

- *однопроходный диапазон* допускает однократную прямую итерацию;
- *прямой диапазон* допускает (неограниченный) проход вперед и является однопроходном диапазоном;
- *двухпроходный диапазон* допускает итерацию вперед и назад и является прямым диапазоном;
- *диапазон произвольного доступа* обеспечивает произвольный доступ к элементу и является двунаправленным диапазоном.

Строковые алгоритмы Boost разработаны для `std::string`, которая удовлетворяет концепции диапазона произвольного доступа. По большей части тот факт, что строковые алгоритмы Boost принимают Boost Range, а не `std::string`, — абсолютно прозрачная абстракция для пользователей. Читая документацию, вы можете мысленно заменить Range строкой.

Предикаты

Строковые алгоритмы Boost активно используют предикаты. Их можно применять напрямую, добавив заголовок `<boost/algorithm/string/predicate. hpp>`. Большинство предикатов, содержащихся в этом заголовке, принимают два диапазона, `r1` и `r2`, и возвращают логическое значение на основе их отношений. Например, предикат `start_with` возвращает `true`, если `r1` начинается с `r2`.

Каждый предикат имеет независимую от регистра версию, которую можно применить, добавив букву `i` к имени метода, например `istarts_with`. В листинге 15.27 приведены методы `starts_with` и `istarts_with`.

Листинг 15.27. И start_with, и istarts_with проверяют начальные символы диапазона

```
#include <string>
#include <boost/algorithm/string/predicate.hpp>

TEST_CASE("boost::algorithm") {
  using namespace boost::algorithm;
  using namespace std::literals::string_literals;
  std::string word("cymotrichous"); ❶
  SECTION("starts_with tests a string's beginning") {
    REQUIRE(starts_with(word, "cymo"s)); ❷
  }
  SECTION("istarts_with is case insensitive") {
    REQUIRE(istarts_with(word, "cYmO"s)); ❸
  }
}
```

В примере инициализируется строка, содержащая `cymotrichous` ❶. Первый тест показывает, что `start_with` возвращает `true` при передаче `word` и `cymo` ❷. Независимая от регистра версия `istarts_with` также возвращает `true` для заданного `word` и `cYmO` ❸.

Обратите внимание на то, что `<boost/algorithm/string/predicate.hpp>` также содержит предикат `all`, который принимает один диапазон `r` и предикат `p`. Он возвращает `true`, если `p` вычисляется как `true` для всех элементов `r`, как показано в листинге 15.28.

Листинг 15.28. Предикат all оценивает, удовлетворяют ли предикаты всем элементам в диапазоне

```
TEST_CASE("boost::algorithm::all evaluates a predicate for all elements") {
  using namespace boost::algorithm;
  std::string word("juju"); ❶
  REQUIRE(all(word❷, [](auto c) { return c == 'j' || c =='u'; }❸));
}
```

Здесь инициализируется строка, содержащая `juju` ❶, которая передается в `all` как диапазон ❷. Передается лямбда-предикат, который возвращает `true` для букв `j` и `u` ❸. Поскольку в `juju` содержатся только эти буквы, все возвращает `true`.

В таблице 15.12 перечислены предикаты, доступные в `<boost/algorithm/string/predicate.hpp>`. В этой таблице `r`, `r1` и `r2` означают строковые диапазоны, а `p` — предикат сравнения элементов.

Таблица 15.12. Предикаты в библиотеке строковых алгоритмов Boost

Предикат	Возвращает true, если
`starts_with(r1, r2, [p])` `istarts_with(r1, r2)`	`r1` начинается с `r2`; `p` используется для сравнения символов
`ends_with(r1, r2, [p])` `iends_with(r1, r2)`	`r1` заканчивается `r2`; `p` используется для сравнения символов

Продолжение ⇩

Таблица 15.12 *(продолжение)*

Предикат	Возвращает `true`, если
`contains(r1, r2, [p])` `icontains(r1, r2)`	`r1` содержит `r2`; `p` используется для сравнения символов
`equals(r1, r2, [p])` `iequals(r1, r2)`	`r1` равен `r2`; `p` используется для сравнения символов
`lexicographical_compare(r1, r2, [p])` `lexicographical_compare(r1, r2)`	`r1` лексикографически меньше `r2`; `p` используется для сравнения символов
`all(r, [p])`	Все элементы `r` возвращают `true` для `p`

Функции, начинающиеся с `i`, не чувствительны к регистру.

Классификаторы

Классификаторы — это предикаты, которые вычисляют некоторые характеристики символа. Заголовок `<boost/attribute/string/classification.hpp>` предлагает генераторы для создания классификаторов. *Генератор* — это функция, не являющаяся членом, которая действует как конструктор. Некоторые генераторы принимают аргументы для настройки классификатора.

ПРИМЕЧАНИЕ

Конечно, можно так же легко создавать свои собственные предикаты с помощью собственных объектов функций, таких как лямбды, но Boost уже предоставляет меню готовых классификаторов.

Например, генератор `is_alnum` создает классификатор, который определяет, является ли символ буквенно-цифровым. В листинге 15.29 показано, как использовать этот классификатор независимо или вместе с `all`.

Здесь создается `classifier` из генератора `is_alnum` ❶. Первый тест использует `classifier` для вычисления того, что `a` является буквенно-цифровым символом ❷, а `$` — нет ❸. Поскольку все классификаторы являются предикатами, которые работают с символами, можно использовать их вместе со всеми предикатами, рассмотренными в предыдущем разделе, чтобы определить, что `nostarch` содержит все буквенно-цифровые символы ❹, а `@nostarch` — нет ❺.

Листинг 15.29. Генератор is_alum определяет, является ли символ буквенно-цифровым

```
#include <boost/algorithm/string/classification.hpp>

TEST_CASE("boost::algorithm::is_alnum") {
  using namespace boost::algorithm;
  const auto classifier = is_alnum(); ❶
  SECTION("evaluates alphanumeric characters") {
```

```
      REQUIRE(classifier('a')); ❷
      REQUIRE_FALSE(classifier('$')); ❸
    }
    SECTION("works with all") {
      REQUIRE(all("nostarch", classifier)); ❹
      REQUIRE_FALSE(all("@nostarch", classifier)); ❺
    }
  }
```

В таблице 15.13 перечислены классификации символов, доступные в `<boost/algorithm/string/classification.hpp>`. В этой таблице `r` означает диапазон строк, а `beg` и `end` — символы, определяющие диапазон.

Таблица 15.13. Предикаты символов в библиотеке строковых алгоритмов Boost

Предикат	Возвращает `true`, если это ...
`is_space`	Пробел
`is_alnum`	Буквенно-цифровой символ
`is_alpha`	Алфавитный символ
`is_cntrl`	Символ управления
`is_digit`	Десятичная цифра
`is_graph`	Графический символ
`is_lower`	Строчный символ
`is_print`	Печатный символ
`is_punct`	Знак пунктуации
`is_upper`	Прописной символ
`is_xdigit`	Шестнадцатеричная цифра
`is_any_of(r)`	Содержится в `r`
`is_from_range(beg, end)`	Содержится в диапазоне от `beg` до `end`

Искатели

Искатель — это концепт, который определяет положение в диапазоне, соответствующем некоторым заданным критериям, обычно предикату или регулярному выражению. Строковые алгоритмы Boost предоставляют некоторые генераторы для создания искателей в заголовке `<boost/attribute/string/finder.hpp>`.

Например, генератор `nth_finder` принимает диапазон `r` и индекс `n`, и он создает искатель, который будет искать диапазон (взятый как итераторы `begin` и `end`) для `n`-го вхождения `r`, как показано в листинге 15.30.

Листинг 15.30. Генератор nth_finder создает искатель, который находит n-е вхождение последовательности

```
#include <boost/algorithm/string/finder.hpp>

TEST_CASE("boost::algorithm::nth_finder finds the nth occurrence") {
  const auto finder = boost::algorithm::nth_finder("na", 1); ❶
  std::string name("Carl Brutananadilewski"); ❷
  const auto result = finder(name.begin(), name.end()); ❸
  REQUIRE(result.begin() == name.begin() + 12); ❹ // Brutana(n)adilewski
  REQUIRE(result.end() == name.begin() + 14); ❺ // Brutanana(d)ilewski
}
```

Генератор `nth_finder` используется, чтобы создать искатель, который найдет второй экземпляр `na` в диапазоне (`n` — ноль) ❶. Затем создается `name`, содержащий `Carl-Brutananadilewski` ❷, и вызывается `finder` с итераторами `begin` и `end` в `name` ❸. Результатом является диапазон, `begin` которого указывает на второй `n` в `Brutananadilewski` ❹, а `end` которого указывает на первый `d` в `Brutananadilewski` ❺.

В таблице 15.14 перечислены искатели, доступные в `<boost/algorithm/string/finder.hpp>`. В этой таблице `s` означает строку, `p` — предикат сравнения элемента, `n` — целое значение, `beg` и `end` — итераторы, `rgx` — регулярное выражение, а `r` — диапазон строк.

Таблица 15.14. Искатели в библиотеке строковых алгоритмов Boost

Генератор	Создает искатель, который при вызове возвращает...
`first_finder(s, p)`	Первый элемент, соответствующий `s`, с использованием `p`
`last_finder(s, p)`	Последний элемент, соответствующий `s`, с использованием `p`
`nth_finder(s, p, n)`	`n`-й элемент, соответствующий `s`, с использованием `p`
`head_finder(n)`	Первые `n` элементов
`tail_finder(n)`	Последние `n` элементов
`token_finder(p)`	Символ, соответствующий `p`
`range_finder(r)` `range_finder(beg, end)`	`r` независимо от ввода
`regex_finder(rgx)`	Первую подстроку, соответствующую `rgx`

ПРИМЕЧАНИЕ

Алгоритмы Boost String определяют концепт форматирования, который представляет результаты поиска для алгоритма замены. Эти алгоритмы пригодятся только опытному пользователю. Обратитесь к документации по алгоритмам `find_format` в заголовке `<bost/attribute /string/find_format.hpp>` для получения дополнительной информации.

Алгоритмы изменения

Boost содержит *множество* алгоритмов для изменения строки (диапазона). Между заголовками `<boost/algorithm/string/case_conv.hpp>`, `<boost/algorithm/string/trim.hpp>` и `<boost/algorithm/string/replace.hpp>` существуют алгоритмы для преобразования регистра, удаления пробелов, замены и стирания различными способами.

Например, функция `to_upper` преобразует все буквы строки в верхний регистр. Если нужно сохранить оригинал без изменений, можно использовать функцию `to_upper_copy`, которая будет возвращать новый объект. В листинге 15.31 приведены `to_upper` и `to_upper_copy`.

Листинг 15.31. И to_upper, и to_upper_copy меняют регистр строки

```
#include <boost/algorithm/string/case_conv.hpp>

TEST_CASE("boost::algorithm::to_upper") {
  std::string powers("difficulty controlling the volume of my voice"); ❶
  SECTION("upper-cases a string") {
    boost::algorithm::to_upper(powers); ❷
    REQUIRE(powers == "DIFFICULTY CONTROLLING THE VOLUME OF MY VOICE"); ❸
  }
  SECTION("_copy leaves the original unmodified") {
    auto result = boost::algorithm::to_upper_copy(powers); ❹
    REQUIRE(powers == "difficulty controlling the volume of my voice"); ❺
    REQUIRE(result == "DIFFICULTY CONTROLLING THE VOLUME OF MY VOICE"); ❻
  }
}
```

В примере создается строка с именем `powers` ❶. Первый тест вызывает `to_upper` для `powers` ❷, который изменяет его на месте таким образом, чтобы тот содержал все заглавные буквы ❸. Второй тест использует вариант `_copy` для создания новой строки с именем `result` ❹. На строку `powers` это никак не повлияло ❺, а `result` в итоге содержит версию с заглавными буквами ❻.

Некоторые строковые алгоритмы Boost, такие как `replace_first`, также имеют регистрозависимые версии. Просто добавьте `i`, и сопоставление будет продолжено независимо от регистра. Для таких алгоритмов, как `replace_first`, которые также имеют варианты `_copy`, подойдет любая перестановка (`replace_first`, `ireplace_first`, `replace_first_copy` и `ireplace_first_copy`).

Алгоритм `replace_first` и его варианты принимают входной диапазон `s`, диапазон совпадений `m` и диапазон замены `r` и заменяют первый экземпляр `m` в `s` на `r`. В листинге 15.32 показаны алгоритмы `replace_first` и `i_replace_first`.

Здесь создается строка `publisher`, содержащая `No Starch Press` ❶. Первый тест вызывает `replace_first` с `publisher` в качестве входной строки, `No` в качестве строки совпадения и `Medium` в качестве строки замены ❷. После этого `publisher` будет содержать `Medium Starch Press` ❸. Во втором тесте используется вариант `ireplace_first_copy`, который не учитывает регистр и выполняет копирование. `NO` и `MEDIUM`

передаются как совпадения и заменяют строки ❹ соответственно, а результат будет содержать `MEDIUM Starch Press` ❻, тогда как `publisher` останется неизменным ❺.

Листинг 15.32. Как replace_first, так и i_replace_first заменяют соответствующие последовательности строк

```
#include <boost/algorithm/string/replace.hpp>

TEST_CASE("boost::algorithm::replace_first") {
  using namespace boost::algorithm;
  std::string publisher("No Starch Press"); ❶
  SECTION("replaces the first occurrence of a string") {
    replace_first(publisher, "No", "Medium"); ❷
    REQUIRE(publisher == "Medium Starch Press"); ❸
  }
  SECTION("has a case-insensitive variant") {
    auto result = ireplace_first_copy(publisher, "NO", "MEDIUM"); ❹
    REQUIRE(publisher == "No Starch Press"); ❺
    REQUIRE(result == "MEDIUM Starch Press"); ❻
  }}
```

В таблице 15.15 перечислены многие из алгоритмов изменения, доступных в строковых алгоритмах Boost. В этой таблице `r`, `s`, `s1` и `s2` означают строки; `p` — предикат сравнения элементов; `n` — интегральное значение и `rgx` — регулярное выражение.

Таблица 15.15. Алгоритмы изменения в библиотеке алгоритмов Boost String

Алгоритм	Описание
`to_upper(s)` `to_upper_copy(s)`	Преобразует все буквы в `s` в прописные
`to_lower(s)` `to_lower_copy(s)`	Преобразует все буквы в `s` в строчные

`trim_left_copy_if(s, [p])`
`trim_left_if(s, [p])`
`trim_left_copy(s)`
`trim_left(s)`
	Удаляет начальные пробелы из `s`

`trim_right_copy_if(s, [p])`
`trim_right_if(s, [p])`
`trim_right_copy(s)`
`trim_right(s)`
	Удаляет завершающие пробелы из `s`

`trim_copy_if(s, [p])`
`trim_if(s, [p])`
`trim_copy(s)`
`trim(s)`
	Удаляет начальные и завершающие пробелы из `s`

Алгоритм	Описание

`replace_first(s1, s2, r)`
`replace_first_copy(s1, s2, r)`
`ireplace_first(s1, s2, r)`
`ireplace_first_copy(s1, s2, r)`
	Заменяет первое вхождение `s2` в `s1` на `r`

`erase_first(s1, s2)`
`erase_first_copy(s1, s2)`
`ierase_first(s1, s2)`
`ierase_first_copy(s1, s2)`
	Удаляет первое вхождение `s2` в `s1`

`replace_last(s1, s2, r)`
`replace_last_copy(s1, s2, r)`
`ireplace_last(s1, s2, r)`
`ireplace_last_copy(s1, s2, r)`
	Заменяет последнее вхождение `s2` в `s1` на `r`

`erase_last(s1, s2)`
`erase_last_copy(s1, s2)`
`ierase_last(s1, s2)`
`ierase_last_copy(s1, s2)`
	Удаляет последнее вхождение `s2` в `s1`

`replace_nth(s1, s2, n, r)`
`replace_nth_copy(s1, s2, n, r)`
`ireplace_nth(s1, s2, n, r)`
`ireplace_nth_copy(s1, s2, n, r)`
	Заменяет `n`-е вхождение `s2` в `s1` на `r`

`erase_nth(s1, s2, n)`
`erase_nth_copy(s1, s2, n)`
`ierase_nth(s1, s2, n)`
`ierase_nth_copy(s1, s2, n)`
	Удаляет `n`-е вхождение `s2` в `s1`

`replace_all(s1, s2, r)`
`replace_all_copy(s1, s2, r)`
`ireplace_all(s1, s2, r)`
`ireplace_all_copy(s1, s2, r)`
	Заменяет все вхождения `s2` в `s1` на `r`

`erase_all(s1, s2)`
`erase_all_copy(s1, s2)`
`ierase_all(s1, s2)`
`ierase_all_copy(s1, s2)`
	Удаляет все вхождения `s2` в `s1`

`replace_head(s, n, r)` `replace_head_copy(s, n, r)`	Заменяет первые `n` символов `s` на `r`
`erase_head(s, n)` `erase_head_copy(s, n)`	Стирает первые `n` символов `s`

Продолжение ⇩

Таблица 15.15 *(продолжение)*

Алгоритм	Описание
`replace_tail(s, n, r)` `replace_tail_copy(s, n, r)`	Заменяет последние `n` символов `s` на `r`
`erase_tail(s, n)` `erase_tail_copy(s, n)`	Стирает последние `n` символов `s`
`replace_regex(s, rgx, r)` `replace_regex_copy(s, rgx, r)`	Заменяет первый экземпляр `rgx` в `s` на `r`
`erase_regex(s, rgx)` `erase_regex_copy(s, rgx)`	Стирает первый экземпляр `rgx` в `s`
`replace_all_regex(s, rgx, r)` `replace_all_regex_copy(s, rgx, r)`	Заменяет все экземпляры `rgx` в `s` на `r`
`erase_all_regex(s, rgx)` `erase_all_regex_copy(s, rgx)`	Стирает все экземпляры `rgx` в `s`

Разделение и соединение

Строковые алгоритмы Boost содержат функции для разделения и объединения строк в заголовках `<boost/algorithm/string/split.hpp>`и `<boost/algorithm/string/join.hpp>`.

Чтобы разбить строки, предоставляется функция `split` с STL-контейнером `res`, диапазоном `s` и предикатом `p`. Он будет маркировать диапазон `s`, используя предикат `p`, чтобы определить разделители и вставить результаты в `res`. В листинге 15.33 показана работа функции `split`.

Листинг 15.33. Функция split маркирует строку

```
#include <vector>
#include <boost/algorithm/string/split.hpp>
#include <boost/algorithm/string/classification.hpp>

TEST_CASE("boost::algorithm::split splits a range based on a predicate") {
  using namespace boost::algorithm;
  std::string publisher("No Starch Press"); ❶
  std::vector<std::string> tokens; ❷
  split(tokens, publisher, is_space()); ❸
  REQUIRE(tokens[0] == "No"); ❹
  REQUIRE(tokens[1] == "Starch");
  REQUIRE(tokens[2] == "Press");
}
```

Вооружившись `publisher` ❶, вы создаете вектор `tokens` для хранения результатов ❷. `split` вызывается с `tokens` в качестве контейнера результатов, `publisher` в качестве

диапазона и `is_space` в качестве предиката ❸. Это разбивает `publisher` на части по пробелам. После этого `tokens` будет содержать `No`, `Starch` и `Press`, как ожидалось ❹.

Можно выполнить обратную операцию с помощью функции `join`, которая принимает контейнер STL `seq` и строку-разделитель `sep`. Функция `join` будет связывать каждый элемент `seq` с каждым элементом `sep`.

В листинге 15.34 показана полезность `join` и незаменимость оксфордской запятой.

Листинг 15.34. Функция join соединяет stringtokens, используя разделитель

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

TEST_CASE("boost::algorithm::join staples tokens together") {
  std::vector<std::string> tokens{ "We invited the strippers",
                                   "JFK", "and Stalin." }; ❶
  auto result = boost::algorithm::join(tokens, ", "); ❷
  REQUIRE(result == "We invited the strippers, JFK, and Stalin."); ❸
}
```

Создается экземпляр `vector` под названием `tokens` с тремя объектами `string` ❶. Затем используется `join` для связывания составляющих элементов `tokens` с запятой, за которой следует пробел ❷. В результате получается одна `string`, содержащая составляющие элементы, связанные с запятыми и пробелами ❸.

В таблице 15.16 перечислены многие из алгоритмов разделения/объединения, доступных в `<boost /algorithm/string/split.hpp>` и `<boost/algorithm/string/join.hpp>`. В этой таблице `res`, `s`, `s1`, `s2` и `sep` означают строки; `seq` — диапазон строк; `p` — предикат сравнения элементов; а `rgx` — регулярное выражение.

Поиск

Boost String Algorithms предлагает несколько функций для поиска диапазонов в заголовке `<boost/algorithm/string/find.hpp>`. Это по существу удобные обертки вокруг искателей в таблице 15.8.

Таблица 15.16. Алгоритмы разбиения и объединения в библиотеке строковых алгоритмов Boost

Функция	Описание

`find_all(res, s1, s2)`
`ifind_all(res, s1, s2)`
`find_all_regex(res, s1, rgx)`
`iter_find(res, s1, s2)`
	Находит все экземпляры `s2` или `rgx` в `s1`, записывая каждый в `res`

`split(res, s, p)`
`split_regex(res, s, rgx)`
`iter_split(res, s, s2)`
	Разделяет `s`, используя `p`, `rgx` или `s2`, записывая токены в `res`

Продолжение ⇩

Таблица 15.16 *(продолжение)*

Функция	Описание
`join(seq, sep)`	Возвращает строку, соединяющую `seq`, используя `sep` в качестве разделителя
`join_if(seq, sep, p)`	Возвращает строку, соединяющую все элементы `seq`, соответствующие `p`, используя `sep` в качестве разделителя

Например, функция `find_head` принимает диапазон `s` и длину `n` и возвращает диапазон, содержащий первые `n` элементов из `s`. В листинге 15.35 показана работа функции `find_head`.

Листинг 15.35. Функция find_head создает диапазон от начала строки

```
#include <boost/algorithm/string/find.hpp>

TEST_CASE("boost::algorithm::find_head computes the head") {
  std::string word("blandishment"); ❶
  const auto result = boost::algorithm::find_head(word, 5); ❷
  REQUIRE(result.begin() == word.begin()); ❸ // (b)landishment
  REQUIRE(result.end() == word.begin()+5); ❹ // bland(i)shment
}
```

Создается строка `word`, содержащая `blandishment` ❶. Она передается в `find_head` вместе с аргументом длины 5 ❷. `begin` в `result` указывает на начало `word` ❸, а его `end` указывает на следующий элемент после пятого ❹.

В таблице 15.17 перечислены многие из алгоритмов поиска, доступных в `<boost/attribute/string/find.hpp>`. В этой таблице `s`, `s1` и `s2` означают строки; `p` — предикат сравнения элементов; `rgx` — регулярное выражение; `n` — интегральное значение, а `fnd` — средство поиска.

Таблица 15.17. Поиск алгоритмов в библиотеке строковых алгоритмов Boost

Предикат	Ищет...
`find_first(s1, s2)` `ifind_first(s1, s2)`	Первый экземпляр `s2` в `s1`
`find_last(s1, s2)` `ifind_last(s1, s2)`	Последний экземпляр `s2` в `s1`
`find_nth(s1, s2, n)` `ifind_nth(s1, s2, n)`	`n`-й экземпляр `s2` в `s1`
`find_head(s, n)`	Первые `n` символов `s`
`find_tail(s, n)`	Последние `n` символов `s`
`find_token(s, p)`	Первое совпадение символов `p` в `s`
`find_regex(s, rgx)`	Первая подстрока, соответствующая `rgx` в `s`
`find(s, fnd)`	Результат применения `fnd` к `s`

Boost Tokenizer

Boost Tokenizer `boost::tokenizer` — это шаблон класса, который выдает представление ряда токенов, содержащихся в строке. `tokenizer` принимает три необязательных параметра шаблона: функцию токенизатора, тип итератора и тип строки.

Функция токенизатора — это предикат, который определяет, является ли символ разделителем (возвращает `true`) или нет (возвращает `false`). Функция токенизатора по умолчанию интерпретирует пробелы и знаки пунктуации как разделители. Если нужно явно указать разделители, используйте класс `boost::char_separator<char>`, который принимает C-строку, содержащую все символы-разделители. Например, `boost::char_separator<char>(";|,")` разделяется на точку с запятой (;), вертикальную черту (|) и запятую (,).

Тип итератора и тип строки соответствуют типу `string`, которую нужно разделить. По умолчанию это `std::string::const_iterator` и `std::string` соответственно.

Так как `tokenizer` не выделяет память, а `boost::algorithm::split` делает это, настоятельно рекомендуется использовать первый, когда токены `string` нужно перебрать только один раз.

Шаблон `tokenizer` предоставляет методы `begin` и `end`, которые возвращают итераторы ввода, поэтому можно рассматривать его как диапазон значений, соответствующий базовой последовательности токенов.

Листинг 15.36 токенизирует традиционный палиндром `A man, a plan, a canal, Panama!` через запятую.

Листинг 15.36. boost::tokenizer разделяет строки по указанным разделителям

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

TEST_CASE("boost::tokenizer splits token-delimited strings") {
  std::string palindrome("A man, a plan, a canal, Panama!"); ❶
  boost::char_separator<char> comma{ "," }; ❷
  boost::tokenizer<boost::char_separator<char>> tokens{ palindrome, comma }; ❸
  auto itr = tokens.begin(); ❹
  REQUIRE(*itr == "A man"); ❺
  itr++; ❻
  REQUIRE(*itr == " a plan");
  itr++;
  REQUIRE(*itr == " a canal");
  itr++;
  REQUIRE(*itr == " Panama!");
}
```

Здесь создается `palindrome` ❶, `char_separator` ❷ и соответствующий `tokenizer` ❸. Затем итератор извлекается из токенизатора, используя его метод `begin` ❹. Можно обращаться с полученным итератором как обычно, разыменовывая его значение ❺ и увеличивая до следующего элемента ❻.

Локализация

Региональная настройка (*локаль*) — это класс для кодирования культурных предпочтений. Концепт локали обычно кодируется в любой операционной среде, в которой работает приложение. Он также управляет многими настройками, такими как сравнение строк; дата и время, денежное и числовое форматирование; почтовые индексы и номера телефонов.

STL предлагает класс `std::locale` и множество вспомогательных функций и классов в заголовке `<locale>`.

В целях экономии места подробно рассматривать локали мы не будем.

Итоги

В этой главе подробно описан `std::string` и его экосистема. Изучив его сходство с `std::vector`, вы узнали о его встроенных методах обработки данных на человеческом языке, таких как сравнение, добавление, удаление, замена и поиск. Вы посмотрели, как функции преобразования чисел позволяют преобразовывать числа и строки, и изучили роль, которую `std::string_view` играет в передаче строк в программах. Вы также узнали, как использовать регулярные выражения для выполнения сложного сопоставления, поиска и замены на основе потенциально сложных шаблонов. Наконец, вы прошли через библиотеку Boost String Algorithms, которая дополняет и расширяет встроенные методы `std::string` дополнительными методами для поиска, замены, обрезки, стирания, разделения и объединения.

Упражнения

15.1. Рефакторинг программы из листингов 9.30 и 9.31 для использования `std::string`. Создайте строку из ввода программы и измените `AlphaHistogram`, чтобы он принимал `string_view` или `const string&` в своем методе `ingest`. Используйте цикл `for`, основанный на диапазоне, для перебора входящих в строку элементов. Замените тип поля `counts` на ассоциативный контейнер.

15.2. Реализуйте программу, которая определяет, является ли пользовательский ввод палиндромом.

15.3. Реализуйте программу, которая подсчитывает количество гласных в пользовательском вводе.

15.4. Реализуйте программу калькулятора, которая поддерживает сложение, вычитание, умножение и деление любых двух чисел. Попробуйте использовать метод поиска `std::string` и функции преобразования чисел.

15.5. Расширьте программу калькулятора несколькими из следующих способов: добавьте операции умножения или оператор деления по модулю и примите числа с плавающей точкой или круглые скобки.

15.6. Необязательно: узнайте больше о локалях в [localization].

Что еще почитать?

- Международный стандарт ИСО/МЭК (2017) — Язык программирования C++ (Международная организация по стандартизации; Женева, Швейцария; isocpp.org/std/the-standard/)
- «Язык программирования C++», 4-е издание, Бьёрн Страуструп (Бином, 2011)
- «The Boost C++ Libraries», 2nd Edition, Boris Schäling (XML Press, 2014)
- «Стандартная библиотека C++. Справочное руководство», 2-е издание, Николай М. Джосаттис (Вильямс, 2017)

16 Потоки

Либо напишите книгу, стоящую чтения, либо сделайте что-то, стоящее написания книги.

Бенджамин Франклин

В этой главе описываются потоки, основная концепция, которая позволяет соединять входы из любого источника и выходы с любым видом назначения с использованием общей структуры. Вы узнаете о классах, которые образуют базовые элементы этой общей платформы, нескольких встроенных средствах и о том, как включать потоки в пользовательские типы.

Потоки

Поток моделирует *поток данных*. В потоке данные расплываются между объектами, и эти объекты могут выполнять произвольную обработку данных. При работе с потоками вывод — это данные, поступающие в поток, а ввод — данные, поступающие из потока. Эти термины отражают потоки с точки зрения пользователя.

В C++ потоки являются основным механизмом для выполнения ввода и вывода (input/output, I/O). Независимо от источника или назначения можно использовать потоки в качестве общего языка для подключения входов к выходам. STL использует наследование классов для кодирования отношений между различными типами потоков. Основные типы в этой иерархии:

- шаблон класса `std::basic_ostream` в заголовке `<ostream>`, который представляет устройство вывода;

- шаблон класса `std::basic_istream` в заголовке `<istream>`, который представляет устройство ввода;
- шаблон класса `std::basic_iostream` в заголовке `<iostream>` для устройств ввода и вывода.

Все три типа потока требуют двух параметров шаблона. Первый соответствует базовому типу данных потока, а второй — типажу типа.

В этом разделе рассматриваются потоки с точки зрения пользователя, а не с точки зрения разработчика библиотеки. Вы поймете интерфейс потоков и узнаете, как взаимодействовать со стандартными операциями ввода-вывода, файлами и строками, используя встроенную поддержку потоков в STL. Если необходимо реализовать новый вид потока (например, для новой библиотеки или фреймворка), потребуются копия стандарта ИСО C++ 17, несколько рабочих примеров и достаточный запас кофе. Ввод/вывод сложен, и вы увидите, что эта трудность отражена во внутренней сложности реализации потока. К счастью, хорошо спроектированный потоковый класс скрывает большую часть этой сложности от пользователей.

Классы потоков

Все потоковые классы STL, с которыми взаимодействуют пользователи, происходят от `basic_istream`, `basic_ostream` или обоих через `basic_iostream`. Заголовки, которые объявляют каждый тип, также предоставляют специализации `char` и `wchar_t` для этих шаблонов, как показано в таблице 16.1. Эти интенсивно используемые специализации особенно полезны при работе с вводом и выводом данных на человеческом языке.

Таблица 16.1. Специализации шаблонов для шаблонов первичного потока

Шаблон	Параметр	Специализация	Заголовок
`basic_istream`	`char`	`istream`	`<istream>`
`basic_ostream`	`char`	`ostream`	`<ostream>`
`basic_iostream`	`char`	`iostream`	`<iostream>`
`basic_istream`	`wchar_t`	`wistream`	`<istream>`
`basic_ostream`	`wchar_t`	`wostream`	`<ostream>`
`basic_iostream`	`wchar_t`	`wiostream`	`<iostream>`

Объекты в таблице 16.1 являются абстракциями, которые вы можете использовать в программах для написания общего кода. Нужно написать функцию, которая записывает вывод в произвольный источник? Если это так, вы можете принять ссылочный параметр `ostream` и не иметь дело со всеми неприятными деталями реализации. (Позже в разделе «Выходные файловые потоки» на с. 633 вы узнаете, как это сделать.)

Часто нужно выполнить ввод-вывод с пользователем (или с программной средой). Глобальные потоковые объекты предоставляют удобную потоковую оболочку, с которой можно работать.

Глобальные объекты потока

STL предоставляет несколько *глобальных объектов потока* в заголовке `<iostream>`, которые оборачивают потоки ввода, вывода и ошибок стандартного ввода, вывода и вывода ошибок. Эти стандартные для реализации потоки являются предварительно связанными каналами между программой и исполняющей средой. Например, в среде рабочего стола стандартный ввод обычно связывается с клавиатурой, а стандартный вывод и вывод ошибок — с консолью.

ПРИМЕЧАНИЕ

Вспомните, что в первой части мы широко использовали `printf` для записи в стандартный вывод.

В таблице 16.2 перечислены глобальные объекты потока, все из которых находятся в пространстве имен `std`.

Таблица 16.2. Глобальные объекты потока

Объект	Тип	Назначение
`cout` `wcout`	`ostream` `wostream`	Вывод, например на дисплей
`cin` `wcin`	`istream` `wistream`	Ввод, например с клавиатуры
`cerr` `wcerr`	`ostream` `wostream`	Вывод ошибки (небуферизованный)
`clog` `wclog`	`ostream` `wostream`	Вывод ошибки (буферизованный)

Итак, как использовать эти объекты? Ну, потоковые классы поддерживают операции, которые можно разделить на две категории.

- **Форматированные операции.** Могут выполнить некоторый препроцессинг своих входных параметров перед выполнением ввода-вывода.
- **Неформатированные операции.** Выполняют ввод/вывод напрямую.

Следующие разделы объясняют каждую из этих категорий по очереди.

Форматированные операции

Весь форматированный ввод/вывод проходит через две функции: *стандартные операторы потока*, `operator<<` и `operator>>`. Вы узнаете их как операторы сдвига

влево и вправо из «Логических операторов» на с. 247. Несколько запутанно то, что потоки перегружают операторы сдвига влево и вправо совершенно несвязанными функциями. Семантическое значение выражения `i <<5` полностью зависит от типа `i`. Если `i` является целочисленным типом, это выражение означает *взять i и сдвинуть биты влево на пять двоичных цифр*. Если `i` не является целочисленным типом, это означает *записать значение 5 в i*. Хотя это обозначенное столкновение кажется неудачным, на практике оно не доставляет особых хлопот. Просто обратите внимание на используемые типы и хорошо протестируйте код.

Потоки вывода перегружают `operator<<`, который называется *оператором вывода* или *вставки*. Шаблон класса `basic_ostream` перегружает оператор вывода для всех основных типов (кроме `void` и `nullptr_t`) и некоторых контейнеров STL, таких как `basic_string`, `complex` и `bitset`. Как стороннему пользователю вам не нужно беспокоиться о том, как эти перегрузки переводят объекты в читаемый вывод.

В листинге 16.1 показано, как использовать оператор вывода для записи различных типов в `cout`.

Листинг 16.1. Использование cout и operator<< для записи в стандартный вывод

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

using namespace std;

int main() {
  bitset<8> s{ "01110011" };
  string str("Crying zeros and I'm hearing ");
  size_t num{ 111 };
  cout << s; ❶
  cout << '\n'; ❷
  cout << str; ❸
  cout << num; ❹
  cout << "s\n"; ❺
}
--------------------------------------------------------------------------------
01110011 ❶❷
Crying zeros and I'm hearing 111s ❸❹❺
```

Оператор вывода `operator<<`используется для записи `bitset` ❶, `char` ❷, `string` ❸, `size_t` ❹ и строкового литерала с нулевым символом в конце ❺ в стандартный вывод через `cout`. Несмотря на то что в консоль записывается пять разных типов, вы никогда не столкнетесь с проблемами сериализации. (Представьте обручи, через которые вам пришлось бы прыгать, чтобы получить `printf` для получения аналогичного результата при данных типах.)

Одна очень приятная особенность стандартных потоковых операторов заключается в том, что они обычно возвращают ссылку на поток. Концептуально перегрузки обычно определяются следующим образом:

```
ostream& operator<<(ostream&, char);
```

Это означает, что можно объединять операторы вывода вместе. Используя эту технику, можно реорганизовать листинг 16.1, чтобы `cout` появился только один раз, как показано в листинге 16.2.

Листинг 16.2. Рефакторинг листинга 16.1 путем объединения операторов вывода

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

using namespace std;

int main() {
  bitset<8> s{ "01110011" };
  string str("Crying zeros and I'm hearing ");
  size_t num{ 111 };
  cout << s << '\n' << str << num << "s\n"; ❶
}
--------------------------------------------------------------------------------
01110011
Crying zeros and I'm hearing 111s ❶
```

Поскольку каждый вызов `<<` возвращает ссылку на выходной поток (здесь `cout`), вы просто объединяете вызовы, чтобы получить идентичный вывод ❶.

Входные потоки перегружают `>>`, который называется *оператором ввода*, или *экстрактором*. Класс `basic_istream` имеет соответствующие перегрузки для оператора ввода для всех тех же типов, что и `basic_ostream`, и опять же как пользователь вы можете в значительной степени игнорировать детали десериализации.

В листинге 16.3 показано, как использовать оператор ввода для чтения двух объектов `double` и `string` из `cin`, а затем вывода результата подразумеваемой математической операции в стандартный вывод.

Листинг 16.3. Программа примитивного калькулятора, использующая cin и operator<< для сбора ввода

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

using namespace std;

int main() {
  double x, y;
  cout << "X: ";
  cin >> x; ❶
  cout << "Y: ";
  cin >> y; ❷

  string op;
  cout << "Operation: ";
  cin >> op; ❸
  if (op == "+") {
```

```
    cout << x + y; ❹
  } else if (op == "-") {
    cout << x - y; ❺
  } else if (op == "*") {
    cout << x * y; ❻
  } else if (op == "/") {
   cout << x / y; ❼
  } else {
    cout << "Unknown operation " << op; ❽
  }
}
```

Здесь собираются два `doublex` ❶ и `y` ❷, за которыми следует строка `op` ❸, которая кодирует требуемую операцию. Используя оператор `if`, можно вывести результат указанной операции для сложения ❹, вычитания ❺, умножения ❻ и деления ❼ или указать пользователю, что `op` неизвестен ❽.

Чтобы использовать программу, введите запрошенные значения в консоли по указанию. Новая строка отправит ввод (как стандартный ввод) в `cin`, как показано в листинге 16.4.

Листинг 16.4. Пример прогона программы из листинга 16.3, которая вычисляет окружность Земли в милях

```
X: 3959 ❶
Y: 6.283185 ❷
Operation: * ❸
24875.1 ❹
```

Вводятся два `double`-объекта: радиус Земли в милях, **`3959`** ❶ и 2π, **`6.283185`** ❷, и указывается умножение * ❸. Результат — окружность Земли в милях ❹. Обратите внимание, что не нужно указывать десятичную точку для целочисленного значения ❶; поток достаточно умен, чтобы знать, что существует неявное десятичное число.

ПРИМЕЧАНИЕ

Вы можете спросить, что произойдет в листинге 16.4, если ввести нечисловую строку для `X` ❶ или `Y` ❷. Поток входит в состояние ошибки, о котором вы узнаете позже в этой главе в разделе «Состояние потока» на с. 619. В состоянии ошибки поток и программа перестают принимать ввод.

Неформатированные операции

При работе с текстовыми потоками обычно нужно использовать форматированные операторы; однако если вы работаете с двоичными данными или пишете код, который требует низкоуровневого доступа к потокам, то наверняка захотите узнать о неформатированных операциях. Неформатированный ввод-вывод включает много деталей. Для краткости в этом разделе представлена сводка соответствующих методов, поэтому, если нужно использовать неформатированные операции, обратитесь к [input.output].

Класс `istream` имеет много неформатированных методов ввода. Эти методы манипулируют потоками на уровне байтов и приведены в таблице 16.3. В этой таблице `is` имеет тип `std::istream<T>`, `s` означает `char*`, `n` является размером потока, `pos` является типом позиции и `d` является разделителем типа `T`.

Таблица 16.3: Неформатированные операции чтения для `istream`

Метод	Описание
`is.get([c])`	Возвращает следующий символ или записывает в символьную ссылку `c`, если тот предоставлен
`is.get(s, n, [d])` `is.getline(s, n, [d])`	Операция `get` считывает до `n` символов в буфер `s`, останавливаясь, если встречает символ новой строки, или `d`, если тот предоставлен. Операция `getline` делает то же самое, за исключением того, что она также читает символ новой строки. Оба пишут завершающий нулевой символ в `s`. Нужно убедиться, что в `s` достаточно места
`is.read(s, n)` `is.readsome(s, n)`	Операция `read` читает до `n` символов в буфер `s`; достижение конца файла является ошибкой. Операция `readsome` делает то же самое, за исключением того, что она не считает конец файла ошибкой
`is.gcount()`	Возвращает количество символов, прочитанных последней неформатированной операцией чтения
`is.ignore()`	Извлекает и отбрасывает один символ
`is.ignore(n, [d])`	Извлекает и удаляет до `n` символов. Если задан `d`, найденные символы `d` игнорируются
`is.peek()`	Возвращает следующий символ для чтения без извлечения
`is.unget()`	Помещает последний извлеченный символ обратно в строку
`is.putback(c)`	Если `c` — последний извлеченный символ, выполняется `unget`. В противном случае устанавливается `badbit`. Объясняется в разделе «Состояние потока»

Выходные потоки имеют следующие неформатированные операции записи, которые управляют потоками на очень низком уровне, как показано в таблице 16.4. В этой таблице `os` имеет тип `std::ostream<T>`, `s` означает `char*`, а `n` — размер потока.

Таблица 16.4. Неформатированные операции записи для `ostream`

Метод	Описание
`os.put(c)`	Записывает `c`
`os.write(s, n)`	Записывает `n` символов из `s` в поток
`os.flush()`	Записывает все буферизованные данные на текущее устройство

Специальное форматирование для основных типов

Все фундаментальные типы, кроме `void` и `nullptr`, имеют перегрузки операторов ввода и вывода, но у некоторых есть специальные правила:

- **`char` и `wchar_t`.** Оператор ввода пропускает пробел при присваивании для символьных типов;
- **`char*` и `wchar_t*`.** Оператор ввода сначала пропускает пробел, а затем читает строку, пока не встретит другой пробел или конец файла (EOF). Нужно зарезервировать достаточно места для ввода;
- **`void*`.** Форматы адресов зависят от реализации операторов ввода и вывода. В настольных системах адреса принимают шестнадцатеричный буквенный формат, например `0x01234567` для 32-разрядных или `0x0123456789abcdef` для 64-разрядных;
- **`bool`.** Операторы ввода и вывода обрабатывают логические значения как числа: `1` для `true` и `0` для `false`;
- **Числовые типы.** Оператор ввода требует, чтобы ввод начинался хотя бы с одной цифры. Некорректно сформированные входные числа дают нулевой результат.

Эти правила на первый взгляд могут показаться немного странными, но они окажутся довольно простыми, как только вы к ним привыкнете.

ПРИМЕЧАНИЕ

Избегайте чтения строк в стиле C, поскольку для этого нужно убедиться, что было выделено достаточно места для входных данных. Неспособность выполнить адекватную проверку приводит к неопределенному поведению и, возможно, к серьезным уязвимостям безопасности. Вместо этого используйте `std::string`.

Состояние потока

Состояние потока указывает, произошел ли сбой ввода-вывода. Каждый тип потока предоставляет постоянные статические члены, которые в совокупности называются его *битами*, указывающими на возможное состояние потока: `goodbit`, `badbit`, `eofbit` и `failbit`. Чтобы определить, находится ли поток в конкретном состоянии, вызываются функции-члены, которые возвращают логическое значение, указывающее, находится ли поток в соответствующем состоянии. В таблице 16.5 перечислены эти функции-члены, состояние потока, соответствующее результату `true`, и значение этого состояния.

ПРИМЕЧАНИЕ

Чтобы сбросить состояние потока для указания хорошего рабочего состояния, можно вызвать его метод `clear()`.

Таблица 16.5. Возможные состояния потока, методы доступа к ним и их значения

Метод	Состояние	Значение
`good()`	`goodbit`	Поток находится в хорошем рабочем состоянии
`eof()`	`eofbit`	Поток достиг конца файла
`fail()`	`failbit`	Операция ввода или вывода завершилась неудачно, но поток все еще может находиться в хорошем рабочем состоянии
`bad()`	`badbit`	Произошла катастрофическая ошибка, и поток не в хорошем состоянии

Потоки реализуют неявное преобразование `bool` (`operator bool`), поэтому можно просто и напрямую проверить, находится ли поток в хорошем рабочем состоянии. Например, можно читать ввод из стандартного ввода слово за словом, пока не встретится EOF (или какое-либо другое условие сбоя), используя простой цикл `while`. В листинге 16.5 приведена простая программа, которая использует эту технику для генерации количества слов из стандартного ввода.

Листинг 16.5. Программа, которая считает слова из стандартного ввода

```
#include <iostream>
#include <string>
int main() {
  std::string word; ❶
  size_t count{}; ❷
  while (std::cin >> word) ❸
    count++; ❹
  std::cout << "Discovered " << count << " words.\n"; ❺
}
```

Объявляется `string` с именем `word` для получения слов из стандартного ввода ❶ и инициализируется переменная `count`, равная нулю ❷. Внутри логического выражения цикла `while` назначается новый вход для слова ❸. Если это удастся, `count` увеличивается ❹. Если нет — например, из-за EOF — увеличение прекращается и выводится итоговый счет ❺.

Можно попробовать два метода для тестирования листинга 16.5. Во-первых, можно просто вызвать программу, ввести некоторые данные и предоставить EOF. Способ отправить EOF зависит от вашей операционной системы. В командной строке Windows можно ввести EOF, нажав `Ctrl-Z` и `Enter`. В Linux Bash или в оболочке OS X нужно нажать `Ctrl-D`. В листинге 16.6 показано, как вызвать листинг 16.5 из командной строки Windows.

Листинг 16.6. Запуск программы из листинга 16.5 с помощью ввода в консоли

```
$ listing_16_5.exe ❶
Size matters not. Look at me. Judge me by my size, do you? Hmm? Hmm. And well
you should not. For my ally is the Force, and a powerful ally it is. Life
creates it, makes it grow. Its energy surrounds us and binds us. Luminous
beings are we, not this crude matter. You must feel the Force around you;
here, between you, me, the tree, the rock, everywhere, yes. ❷
^Z ❸
Discovered 70 words. ❹
```

Сначала вызывается программа ❶. Затем вводится произвольный текст, за которым следует новая строка ❷. Затем вводится EOF. Командная строка Windows показывает несколько загадочную последовательность `^Z` в командной строке, после чего нужно нажать `Enter`. Это заставляет `std::cin` переходить в состояние `eofbit`, завершая цикл `while` в листинге 16.5 ❸. Программа указывает, что в стандартный ввод было отправлено 70 слов ❹.

В Linux и macOS, а также в Windows PowerShell есть другой вариант. Вместо ввода непосредственно в консоль можно сохранить текст в файл, скажем, *yoda.txt*. Хитрость заключается в том, чтобы применить `cat` для чтения текстового файла, а затем использовать вертикальную черту |, чтобы отправить содержимое в программу. Вертикальная черта «направляет» стандартный вывод программы слева в стандартный поток программы справа. Следующая команда демонстрирует этот процесс:

```
$ cat yoda.txt❶ |❷ ./listing_15_4❸
Discovered 70 words.
```

Команда `cat` читает содержимое *yoda.txt* ❶. Вертикальная черта ❷ направляет стандартный вывод `cat` в стандартный вывод `listing_15_4` ❸. Поскольку `cat` отправляет EOF, не нужно вводить вручную конец *yoda.txt*.

Иногда необходимо, чтобы потоки генерировали исключение при возникновении определенных битов сбоя. Можно легко сделать это с помощью метода `exceptions` потока, который принимает один аргумент, соответствующий биту, который должен выбросить исключение. Если нужно несколько битов, можно просто объединить их, используя логическое ИЛИ (|).

В листинге 16.7 показано, как выполнить рефакторинг в листинге 16.5, чтобы он обрабатывал `badbit` с исключениями и `eofbit/failbit` с обработкой по умолчанию.

Листинг 16.7. Рефакторинг листинга 16.5 для обработки badbit с исключениями

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

using namespace std;

int main() {
  cin.exceptions(istream::badbit); ❶
  string word;
  size_t count{};
  try { ❷
    while(cin >> word) ❸
      count++;
    cout << "Discovered " << count << " words.\n"; ❹
  } catch (const std::exception& e) { ❺
    cerr << "Error occurred reading from stdin: " << e.what(); ❻
  }
}
```

Вы запускаете программу, вызывая метод исключений в `std::cin` ❶. Поскольку `cin`— это `istream`, `istream::badbit` передается в качестве аргумента в `exception`, указывая, что нужно, чтобы `cin` генерировал исключение каждый раз при попадании в катастрофическое состояние. Чтобы учесть возможные исключения, существующий код оборачивается в блок `try-catch` ❷, поэтому если `cin` устанавливает `badbit` во время чтения ввода ❸, пользователь никогда не получит сообщение о числе слов ❹. Вместо этого программа перехватывает полученное исключение ❺ и печатает сообщение об ошибке ❻.

Буферизация и сброс

Многие шаблоны классов `ostream` включают вызовы операционной системы, например для записи в консоль, файл или сетевой сокет. По сравнению с другими вызовами функций системные вызовы обычно считаются медленными. Вместо того чтобы вызывать системный вызов для каждого выходного элемента, приложение может подождать несколько элементов, а затем отправить их все вместе для повышения производительности.

Поведение в очереди называется *буферизацией*. Когда поток очищает буферизованный вывод, это называется *сбросом*. Обычно это поведение полностью прозрачно для пользователя, но иногда нужно вручную сбросить `ostream`. Для этого (и других задач) стоит обратиться к манипуляторам.

Манипуляторы

Манипуляторы — это специальные объекты, которые изменяют то, как потоки интерпретируют ввод или формат вывода. Манипуляторы существуют для выполнения многих видов изменений потока. Например, `std::ws` изменяет `istream`, чтобы пропустить пробелы. Вот некоторые другие манипуляторы, которые работают с `ostream`:

- `std::flush` очищает любой буферизованный вывод непосредственно в `ostream`;
- `std::ends` отправляет нулевой байт;
- `std::endl` похож на `std::flush` за исключением того, что он отправляет новую строку перед сбросом.

Таблица 16.6 обобщает манипуляторы в заголовках `<istream>` и `<ostream>`.

Например, можно заменить ❹ в листинге 16.7 следующим:

```
cout << "Discovered " << count << " words." << endl;
```

Будет напечатана новая строка и сброшен вывод.

ПРИМЕЧАНИЕ

Используйте `std::endl`, когда программа на некоторое время закончила вывод текста в поток, и `\n`, когда знаете, что программа скоро выведет больше текста.

Таблица 16.6. Четыре манипулятора в заголовках `<istream>` и `<ostream>`

Манипулятор	Класс	Поведение
`ws`	`istream`	Пропускает все пробелы
`flush`	`ostream`	Записывает любые буферизованные данные в поток, вызывая его метод `flush`
`ends`	`ostream`	Посылает нулевой байт
`endl`	`ostream`	Посылает новую строку и сбрасывает

stdlib предоставляет много других манипуляторов в заголовке `<ios>`. Можно, например, определить, будет ли `ostream` представлять логические значения в текстовом (`boolalpha`) или числовом (`noboolalpha`) формате; целочисленные значения как восьмеричные (`oct`), десятичные (`dec`) или шестнадцатеричные (`hex`); числа с плавающей точкой в виде десятичной записи (фиксированной) или научной записи. Просто передайте один из этих манипуляторов в `ostream` с помощью `operator<<`, и *все* последующие вставки соответствующего типа будут обрабатываться (а не только непосредственно предшествующий операнд).

Также можно установить параметр ширины потока, используя манипулятор `setw`. Параметр ширины потока имеет различные эффекты в зависимости от потока. Например, с помощью `std::cout setw` будет фиксировать количество выходных символов, выделенных для следующего выходного объекта. Кроме того, для вывода с плавающей запятой `setprecision` установит точность следующих чисел.

В листинге 16.8 показано, как эти манипуляторы выполняют функции, аналогичные функциям различных спецификаторов формата `printf`.

Листинг 16.8. Программа, где показаны некоторые манипуляторы, доступные в заголовке <iomanip>

```
#include <iostream>
#include <iomanip>

using namespace std;

int main() {
  cout << "Gotham needs its " << boolalpha << true << " hero."; ❶
  cout << "\nMark it " << noboolalpha << false << "!"; ❷
  cout << "\nThere are " << 69 << "," << oct << 105 << " leaves in here."; ❸
  cout << "\nYabba " << hex << 3669732608 << "!"; ❹
  cout << "\nAvogadro's number: " << scientific << 6.0221415e-23; ❺
  cout << "\nthe Hogwarts platform: " << fixed << setprecision(2) << 9.750123; ❻
  cout << "\nAlways eliminate " << 3735929054; ❼
  cout << setw(4) << "\n"
       << 0x1 << "\n"
       << 0x10 << "\n"
       << 0x100 << "\n"
       << 0x1000 << endl; ❽
}
--------------------------------------------------------------------------------
```

```
Gotham needs its true hero. ❶
Mark it 0! ❷
There are 69,151 leaves in here. ❸
Yabba dabbad00! ❹
Avogadro's Number: 6.022142e-23 ❺
the Hogwarts platform: 9.75 ❻
Always eliminate deadc0de ❼
1
10
100
1000 ❽
```

Манипулятор `boolalpha` в первой строке дает команду логическим значениям выводить текстовые значения как `true` и `false` ❶, тогда как `noboolalpha` заставляет их выводить 1 и 0 вместо этого ❷. Для целочисленных значений можно выводить восьмеричные с `oct` ❸ или шестнадцатеричные представления с `hex` ❹. Для значений с плавающей точкой можно указать научную нотацию с помощью `scientific` ❺, а также установить число цифр для вывода с помощью `setprecision` и указать десятичную нотацию с помощью `fixed` ❻. Поскольку манипуляторы применяются ко всем последующим объектам, которые вставляются в поток, при выводе другого интегрального значения в конце программы применяется последний интегральный манипулятор (`hex`), поэтому получается шестнадцатеричное представление ❼. Наконец, `setw` используется, чтобы установить ширину поля для вывода равной 4, и выводятся некоторые интегральные значения ❽.

В таблице 16.7 сведены распространенные манипуляторы.

Таблица 16.7. Многие манипуляторы, доступные в заголовке `<iomanip>`

Манипулятор	Поведение
`Boolalpha`	Представляет логические значения в текстовом виде, а не в числовом
`Noboolalpha`	Представляет логические значения в числовом виде, а не в текстовом
`Oct`	Представляет целочисленные значения как восьмеричные
`Dec`	Представляет целочисленные значения как десятичные
`Hex`	Представляет целочисленные значения как шестнадцатеричные
`setw(n)`	Устанавливает параметр ширины потока в `n`. Точный результат зависит от потока
`setprecision(p)`	Определяет точность с плавающей точкой как `p`
`Fixed`	Представляет числа с плавающей точкой в десятичной записи
`Scientific`	Представляет числа с плавающей точкой в научной записи

ПРИМЕЧАНИЕ

См. главу 15 в «Стандартной библиотеке C++», 2-е издание, Николая М. Джосаттиса или [iostream.format].

Пользовательские типы

Можно заставить пользовательские типы работать с потоками, реализуя определенные функции, не являющиеся членами. Для реализации оператора вывода типа `YourType` следующее объявление функции реализует большинство целей:

```
ostream&❶ operator<<(ostream&❷ s, const YourType& m ❸);
```

В большинстве случаев вы просто возвращаете ❶ тот же `ostream`, который получаете ❷. Вам решать, как отправить вывод в `ostream`. Но обычно это включает в себя доступ к полям в `YourType` ❸, необязательное выполнение некоторого форматирования и преобразования, а затем использование оператора вывода. Например, в листинге 16.9 показано, как реализовать оператор вывода для `std::vector` для вывода его размера, емкости и элементов.

Листинг 16.9. Программа, где реализован оператор вывода для vector

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

using namespace std;

template <typename T>
ostream& operator<<(ostream& s, vector<T> v) { ❶
  s << "Size: " << v.size()
    << "\nCapacity: " << v.capacity()
    << "\nElements:\n"; ❷
  for (const auto& element : v)
    s << "\t" << element << "\n"; ❸
  return s; ❹
}

int main() {
  const vector<string> characters {
    "Bobby Shaftoe",
    "Lawrence Waterhouse",
    "Gunter Bischoff",
    "Earl Comstock"
  }; ❺
  cout << characters << endl; ❻

  const vector<bool> bits { true, false, true, false }; ❼
  cout << boolalpha << bits << endl; ❽
}
```

```
--------------------------------------------------------------------------------
Size: 4
Capacity: 4
Elements: ❷
Bobby Shaftoe ❸
Lawrence Waterhouse ❸
Gunter Bischoff ❸
```

```
Earl Comstock ❸
Size: 4
Capacity: 32
Elements: ❷
true ❸
false ❸
true ❸
false ❸
```

Сначала определяется пользовательский оператор вывода в качестве шаблона, используя параметр шаблона в качестве параметра шаблона `std::vector` ❶. Это позволяет использовать оператор вывода для многих типов векторов (при условии, что тип T также поддерживает оператор вывода). Первые три строки вывода дают размер и емкость `vector`, а также заголовок `Elements`, указывающий, что за ним следуют элементы вектора ❷. Следующий цикл `for` выполняет итерации по элементам в `vector`, отправляя каждый в отдельной строке в `ostream` ❸. Наконец, возвращается ссылка на поток `s` ❹.

Внутри `main` инициализируется вектор `characters`, содержащий четыре `string` ❺. Благодаря пользовательскому оператору вывода можно просто отправлять символы в `cout`, как если бы это был базовый тип ❻. Во втором примере используется `vector<bool>` с названием `bits`, который также инициализируется четырьмя элементами ❼ и выводится в стандартный вывод ❽. Обратите внимание, что используется манипулятор `boolalpha`, поэтому при запуске пользовательского оператора вывода элементы `bool` выводятся в виде текста ❸.

Также можно предоставить пользовательские операторы ввода, которые работают аналогично. Для этого определяется следующий формат:

```
istream&❶ operator>>(istream&❷ s, YourType& m ❸);
```

Как и в случае с оператором вывода, оператор ввода обычно возвращает ❶ тот же поток, который получает ❷. Однако, в отличие от оператора вывода, ссылка на `YourType` обычно не будет `const`, потому что нужно будет изменить соответствующий объект, используя входные данные из потока ❸.

В листинге 16.10 показано, как задать оператор ввода для очереди, чтобы он помещал элементы в контейнер до тех пор, пока не произойдет сбой вставки (например, из-за символа EOF).

Пользовательский оператор ввода является шаблоном функции, поэтому можно принять любую очередь, содержащую тип, который поддерживает оператор ввода ❶. Во-первых, создается элемент типа `T`, чтобы можно было хранить входные данные из `istream` ❷. Далее используется знакомая конструкция `while`, чтобы принимать входные данные из `istream`, пока операция ввода не завершится неудачей ❸. (Вспомните из раздела «Состояние потока», что потоки могут переходить в состояния отказа во многих случаях, включая достижение EOF или ошибку ввода-вывода.) После каждой

вставки результат перемещается в `emplace_back` в очереди, чтобы избежать ненужных копий ❹. После завершения вставки просто возвращается ссылка на `istream` ❺.

Листинг 16.10. Программа с реализацией оператора ввода для очереди

```
#include <iostream>
#include <deque>

using namespace std;

template <typename T>
istream& operator>>(istream& s, deque<T>& t) { ❶
  T element; ❷
  while (s >> element) ❸
    t.emplace_back(move(element)); ❹
  return s; ❺
}

int main() {
  cout << "Give me numbers: "; ❻
  deque<int> numbers;
  cin >> numbers; ❼
  int sum{};
  cout << "Cumulative sum:\n";
  for(const auto& element : numbers) {
    sum += element;
    cout << sum << "\n"; ❽
  }
}
--------------------------------------------------------------------------------
Give me numbers: ❻ 1 2 3 4 5 ❼
Cumulative sum:
1 ❽
3 ❽
6 ❽
10 ❽
15 ❽
```

В `main` у пользователя запрашиваются числа ❻, а затем используется оператор вставки во вновь инициализированной очереди для вставки элементов из стандартного ввода. В этом примере программы вы вводите цифры от 1 до 5 ❼. Забавы ради вычисляется общая сумма путем перебора каждого элемента и сохранения счета и выводится результат этой итерации ❽.

ПРИМЕЧАНИЕ

Предыдущие примеры — это простые пользовательские реализации операторов ввода и вывода. Возможно, нужно будет разработать эти реализации в производственном коде. Например, реализации работают только с классами `ostream`, что означает, что они не будут работать с любыми несимвольными последовательностями.

Строковые потоки

Классы строковых потоков предоставляют возможности для чтения и записи в последовательности символов. Эти классы полезны в нескольких ситуациях. Входные строки особенно полезны, если нужно разбить строковые данные на типы. Поскольку можно использовать оператор ввода, доступны все стандартные возможности манипулятора. Выходные строки отлично подходят для построения строк из ввода переменной длины.

Выходные строковые потоки

Выходные строковые потоки предоставляют семантику выходного потока для символьных последовательностей, и все они происходят от шаблона класса `std::basic_ostringstream` в заголовке `<sstream>`, который обеспечивает следующие специализации:

```
using ostringstream = basic_ostringstream<char>;
using wostringstream = basic_ostringstream<wchar_t>;
```

Выходные строковые потоки поддерживают все те же функции, что и `ostream`. Всякий раз при отправке входных данных в поток строк поток сохраняет эти входные данные во внутреннем буфере. Можно представить это как функционально эквивалентную операцию добавления строки (за исключением того, что строковые потоки потенциально более эффективны).

Выходные строковые потоки также поддерживают метод `str()`, который имеет два режима работы. Если аргумента нет, `str` возвращает копию внутреннего буфера как `basic_string` (поэтому `ostringstream` возвращает строку; `wostringstream` возвращает `wstring`). При наличии одного аргумента `basic_string` строковый поток заменит текущее содержимое своего буфера содержимым аргумента. В листинге 16.11 показано, как использовать поток `ostring`, посылать ему символьные данные, создавать строку, сбрасывать ее содержимое и повторять эти действия.

Листинг 16.11. Использование ostringstream для построения строк

```
#include <string>
#include <sstream>

TEST_CASE("ostringstream produces strings with str") {
  std::ostringstream ss; ❶
  ss << "By Grabthar's hammer, ";
  ss << "by the suns of Worvan. ";
  ss << "You shall be avenged."; ❷
  const auto lazarus = ss.str(); ❸

  ss.str("I am Groot."); ❹
  const auto groot = ss.str(); ❺

  REQUIRE(lazarus == "By Grabthar's hammer, by the suns"
                     " of Worvan. You shall be avenged.");
```

```
  REQUIRE(groot == "I am Groot.");
}
```

После объявления `ostringstream` ❶ он обрабатывается как любой другой `ostream` и используется оператор вывода для отправки ему трех отдельных последовательностей символов ❷. Далее вызывается `str` без аргумента, который создает строку с именем `lazarus` ❸. Затем вызывается `str` со строковым литералом `I am Groot` ❹, который заменяет содержимое `ostringstream` ❺.

ПРИМЕЧАНИЕ

Вспомните из раздела «Строки в стиле C» на с. 103, что можно поместить несколько строковых литералов в последовательные строки, и компилятор будет рассматривать их как один. Это сделано исключительно в целях форматирования исходного кода.

Входные строковые потоки

Входные строковые потоки предоставляют семантику входного потока для символьных последовательностей, и все они происходят от шаблона класса `std::basic_istringstream` в заголовке `<sstream>`, который обеспечивает следующие специализации:

```
using istringstream = basic_istringstream<char>;
using wistringstream = basic_istringstream<wchar_t>;
```

Они аналогичны специализациям `basic_ostringstream`. Можно создавать входные строковые потоки, передавая `basic_string` с соответствующей специализацией (строка для `istringstream` и `wstring` для `wistringstream`). Листинг 16.12 показывает построение входного строкового потока со строкой, содержащей три числа, и использование оператора ввода для их извлечения. (Вспомните из раздела «Форматированные операции» на с. 614, что пробел является подходящим разделителем для строковых данных.)

Листинг 16.12. Использование строки для создания объектов istringstream и извлечения числовых типов

```
TEST_CASE("istringstream supports construction from a string") {
  std::string numbers("1 2.23606 2"); ❶
  std::istringstream ss{ numbers }; ❷
  int a;
  float b, c, d;
  ss >> a; ❸
  ss >> b; ❹
  ss >> c;
  REQUIRE(a == 1);
  REQUIRE(b == Approx(2.23606));
  REQUIRE(c == Approx(2));
  REQUIRE_FALSE(ss >> d); ❺
}
```

`string` создается из литерала 1 2.23606 2 ❶, который передается в конструктор потока `istringstreamss` ❷. Это позволяет использовать оператор ввода для анализа объектов `int` ❸ и `float` ❹, как и любой другой поток ввода. Как только поток исчерпан и оператор вывода потерпел неудачу, `ss` преобразуется в `false` ❺.

Строковые потоки, поддерживающие ввод и вывод

Кроме того, если нужен строковый поток, который поддерживает операции ввода и вывода, можно использовать `basic_stringstream`, который имеет следующие специализации:

```
using stringstream = basic_stringstream<char>;
using wstringstream = basic_stringstream<wchar_t>;
```

Этот класс поддерживает операторы ввода и вывода, метод `str` и создание из строки. В листинге 16.13 показано, как использовать комбинацию операторов ввода и вывода для извлечения токенов из строки.

Листинг 16.13. Использование потока строк для ввода и вывода

```
TEST_CASE("stringstream supports all string stream operations") {
  std::stringstream ss;
  ss << "Zed's DEAD"; ❶

  std::string who;
  ss >> who; ❷
  int what;
  ss >> std::hex >> what; ❸

  REQUIRE(who == "Zed's");
  REQUIRE(what == 0xdead);
}
```

В коде создается `stringstream` и отправляется `Zed's DEAD` с оператором вывода ❶. Затем анализируется `Zed` из `stringstream`, используя оператор ввода ❷. Поскольку `DEAD` является допустимым шестнадцатеричным целым числом, используется оператор ввода и манипулятор `std::hex`, чтобы извлечь его в `int` ❸.

ПРИМЕЧАНИЕ

Все строковые потоки являются перемещаемыми.

Сводка операций строковых потоков

В таблице 16.8 приведен неполный список операций `basic_stringstream`. В этой таблице `ss`, `ss1` и `ss2` имеют тип `std::basic_stringstream<T>`; `s` означает `std::basic_string<T>`; `obj` — отформатированный объект; `pos` — тип позиции; `dir` — `std::ios_base::seekdir`; и `flg` — `std::ios_base::iostate`.

Таблица 16.8. Неполный список операций `std::basic_stringstream`

Операция	Примечания
`basic_stringstream<T> { [s], [om] }`	Выполняет фигурную инициализацию только что созданной строки. По умолчанию пустая строка `s` и открытый режим ввода|вывода `om`
`basic_stringstream<T> { move(ss) }`	Принимает владение внутренним буфером `ss`
`~basic_stringstream`	Уничтожает внутренний буфер
`ss.rdbuf()`	Возвращает простой строковый объект устройства
`ss.str()`	Получает содержимое строкового объекта устройства
`ss.str(s)`	Устанавливает содержимое строкового объекта устройства в `s`
`ss >> obj`	Извлекает отформатированные данные из потока строк
`ss << obj`	Вставляет отформатированные данные в поток строк
`ss.tellg()`	Возвращает индекс позиции ввода
`ss.seekg(pos)` `ss.seekg(pos, dir)`	Устанавливает индикатор положения ввода
`ss.flush()`	Синхронизирует используемое устройство

`ss.good()`
`ss.eof()`
`ss.bad()`
`!ss`
	Проверяет биты потока строк

`ss.exceptions(flg)`	Настраивает поток строк для выдачи исключения всякий раз, когда устанавливается бит в `flg`
`ss1.swap(ss2)` `swap(ss1, ss2)`	Заменяет все элементы `ss1` на элементы `ss2`

Файловые потоки

Классы файлового потока предоставляют средства для чтения и записи в последовательности символов. Структура класса файлового потока соответствует структуре классов строкового потока. Шаблоны классов файлового потока доступны для ввода, вывода и того и другого вместе.

Классы файлового потока предоставляют следующие основные преимущества по сравнению с использованием собственных системных вызовов для взаимодействия с содержимым файла:

- вы получаете обычные потоковые интерфейсы, которые предоставляют богатый набор функций для форматирования и управления выводом;
- классы файлового потока являются оболочками RAII для файлов, что означает невозможность утечки ресурсов, таких как файлы;
- классы файлового потока поддерживают семантику переноса, поэтому можно жестко контролировать размещение файлов.

Открытие файлов с помощью потоков

Есть два варианта открытия файла с любым файловым потоком. Первый вариант — это метод `open`, который принимает `const char* filename`, и необязательный аргумент битовой маски `std::ios_base::openmode`. Аргумент `openmode` может быть одной из многих возможных комбинаций значений, перечисленных в таблице 16.9.

Таблица 16.9. Возможные состояния потока, методы доступа и их значения

Флаг (в `std::ios`)	Файл	Значение
`In`	Должен существовать	Чтение
`Out`	Создается, если не существует	Стирание файла; затем запись
`App`	Создается, если не существует	Добавление
`in|out`	Должен существовать	Чтение и запись с начала
`in|app`	Создается, если не существует	Обновление с конца
`out|app`	Создается, если не существует	Добавление
`out|trunc`	Создается, если не существует	Стирание файла; затем чтение и запись
`in|out|app`	Создается, если не существует	Обновление с конца
`in|out|trunc`	Создается, если не существует	Стирание файла; затем чтение и запись

Кроме того, можно добавить флаг `binary` к любой из этих комбинаций, чтобы перевести файл в *двоичный режим*. В двоичном режиме поток не будет преобразовывать специальные последовательности символов, такие как конец строки (например, возврат каретки плюс перевод строки в Windows) или EOF.

Второй вариант указания файла для открытия — использовать конструктор потока. Каждый файловый поток предоставляет конструктор, принимающий те же аргументы, что и метод `open`. Все классы файловых потоков являются оболочками RAII вокруг своих файловых дескрипторов, поэтому файлы будут автоматически очищаться при разрушении файлового потока. Также можно вручную вызвать метод `close`, который не принимает аргументов. Возможно, вы захотите сделать это, если знаете, что файл больше не нужен, но код написан таким образом, что объект класса файлового потока некоторое время не будет разрушаться.

Файловые потоки также имеют конструкторы по умолчанию, которые не открывают никаких файлов. Чтобы проверить, открыт ли файл, вызовите метод `is_open`, который не принимает аргументов и возвращает логическое значение.

Выходные файловые потоки

Выходные файловые потоки предоставляют семантику выходного потока для символьных последовательностей, и все они происходят от шаблона класса `std::basic_ofstream` в заголовке `<fstream>`, который обеспечивает следующие специализации:

```
using ofstream = basic_ofstream<char>;
using wofstream = basic_ofstream<wchar_t>;
```

Конструктор `basic_ofstream` по умолчанию не открывает файл, а вторым необязательным аргументом конструктора не по умолчанию является `ios::out`.

Всякий раз, когда входные данные отправляются в файловый поток, поток записывает данные в соответствующий файл. В листинге 16.14 показано, как использовать `ofstream` для записи простого сообщения в текстовый файл.

Листинг 16.14. Программа, открывающая файл lunchtime.txt и добавляющая в него сообщение. (Вывод соответствует содержимому lunchtime.txt после выполнения одной программы.)

```
#include <fstream>

using namespace std;

int main() {
  ofstream file{ "lunchtime.txt", ios::out|ios::app }; ❶
  file << "Time is an illusion." << endl; ❷
  file << "Lunch time, " << 2 << "x so." << endl; ❸
}
-------------------------------------------------------------------------------
lunchtime.txt:
Time is an illusion. ❷
Lunch time, 2x so. ❸
```

Инициализируется выходной файловый поток `file` с путем `lunchtime.txt` и флагами `out` и `app` ❶. Поскольку эта комбинация флагов добавляет вывод, любые данные, которые отправляются через оператор вывода в этот файловый поток, добавляются в конец файла. Как и ожидалось, файл содержит сообщение, которое было передано оператору вывода ❷❸.

Благодаря флагу `ios::app` программа добавит вывод в *lunchtime.txt,* если он существует. Например, если снова запустить программу, получится следующий вывод:

```
Time is an illusion.
Lunch time, 2x so.
Time is an illusion.
Lunch time, 2x so.
```

Вторая итерация программы добавила ту же фразу в конец файла.

Входные файловые потоки

Входные файловые потоки обеспечивают семантику входного потока для символьных последовательностей, и все они происходят от шаблона класса `std::basic_ifstream` в заголовке `<fstream>`, который обеспечивает следующие специализации:

```
using ifstream = basic_ifstream<char>;
using wifstream = basic_ifstream<wchar_t>;
```

Конструктор `basic_ifstream` по умолчанию не открывает файл, а вторым необязательным аргументом конструктора не по умолчанию является `ios::in`.

Всякий раз при чтении из файлового потока поток читает данные из соответствующего файла. Рассмотрим следующий пример файла *numbers.txt*:

```
-54
203
9000
0
99
-789
400
```

Листинг 16.15 содержит программу, которая использует `ifstream` для чтения из текстового файла, содержащего целые числа, и возврата максимального значения.

Выходные данные соответствуют вызову программы и передаче пути к файлу *numbers.txt*.

Листинг 16.15. Программа, которая читает текстовый файл numbers.txt и печатает его максимальное целое число

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

using namespace std;

int main() {
  ifstream file{ "numbers.txt" }; ❶
  auto maximum = numeric_limits<int>::min(); ❷
  int value;
  while (file >> value) ❸
    maximum = maximum < value ? value : maximum; ❹
  cout << "Maximum found was " << maximum << endl; ❺
}
--------------------------------------------------------------------------------
Maximum found was 9000 ❺
```

Сначала инициализируется `istream`, чтобы открыть текстовый файл *numbers.txt* ❶. Затем максимальная переменная инициализируется минимальным значением, которое может принимать `int` ❷. Используя идиоматический входной поток и комбинацию цикла `while` ❸, каждое целое число в файле просматривается в цикле,

обновляя максимальное значение при обнаружении более высоких значений ❹. Как только поток файла не может проанализировать больше целых чисел, результат выводится в стандартный вывод ❺.

Обработка ошибок

Как и в случае с другими потоками, файловые потоки завершаются с ошибкой без каких-либо уведомлений. Если используется конструктор файлового потока для открытия файла, нужно проверить метод `is_open`, чтобы определить, успешно ли поток открыл файл. Этот дизайн отличается от большинства других объектов stdlib, где инварианты применяются как исключения. Трудно сказать, почему разработчики библиотек выбрали этот подход, но дело в том, что можно довольно легко выбрать подход, основанный на исключениях.

Можно создать свои собственные фабричные функции для обработки ошибок открытия файлов с исключениями. В листинге 16.16 показано, как реализовать фабрику `ifstream` с именем `open`.

Листинг 16.16. Фабричная функция для генерации ifstream, которая обрабатывает ошибки с генерацией исключений

```
#include <fstream>
#include <string>

using namespace std;

ifstream❶ open(const char* path❷, ios_base::openmode mode = ios_base::in❸) {
  ifstream file{ path, mode }; ❹
  if(!file.is_open()) { ❺
    string err{ "Unable to open file " };
    err.append(path);
    throw runtime_error{ err }; ❻
  }
  file.exceptions(ifstream::badbit);
  return file; ❼
}
```

Фабричная функция возвращает `ifstream` ❶ и принимает те же аргументы, что и конструктор файлового потока (и метод `open`): `path` файла ❷ и `openmode` ❸. Эти два аргумента передаются в конструктор `ifstream` ❹, а затем определяется, был ли файл успешно открыт ❺. Если это не так, выбрасывается `runtime_error` ❻. Если это так, вы даете команду результирующему `ifstream` генерировать исключение всякий раз, когда его `badbit` будет установлен в будущем ❼.

Сводка операций файлового потока

В таблице 16.10 приведен неполный список операций `basic_fstream`. В этой таблице `fs`, `fs1` и `fs2` имеют тип `std::basic_fstream<T>`; `p` — это строка в стиле C, `std::string` или `std::filesystem::path`; `om` — `std::ios_base::openmode`; `s` — `std::basic_string<T>`;

`obj` — отформатированный объект; `pos` — тип позиции; `dir` — `std::ios_base::seekdir`; и `flg` — `std::ios_base::iostate`.

Таблица 16.10. Неполный список операций `std::basic_fstream`

Операция	Примечания
`basic_fstream<T> { [p], [om] }`	Выполняет фигурную инициализацию только что созданного файлового потока. Если `p` указан, пытается открыть файл по пути `p`. По умолчанию не открыт и находится в открытом режиме ввода|вывода
`basic_fstream<T> { move(fs) }`	Принимает владение внутренним буфером `fs`
`~basic_fstream`	Уничтожает внутренний буфер
`fs.rdbuf()`	Возвращает простой файловый объект устройства
`fs.str()`	Получает содержимое файлового объекта устройства
`fs.str(s)`	Устанавливает содержимое строкового объекта устройства в `s`
`fs >> obj`	Извлекает отформатированные данные из файлового потока
`fs << obj`	Вставляет отформатированные данные в файловый поток
`fs.tellg()`	Возвращает индекс позиции ввода
`fs.seekg(pos)` `fs.seekg(pos, dir)`	Устанавливает индикатор положения ввода
`fs.flush()`	Синхронизирует используемое устройство

`fs.good()`
`fs.eof()`
`fs.bad()`
`!fs`
	Проверяет биты файлового потока

`fs.exceptions(flg)`	Настраивает поток строк для выдачи исключения всякий раз, когда устанавливается бит в `flg`
`fs1.swap(fs2)` `swap(fs1, fs2)`	Заменяет все элементы `fs1` на элементы `fs2`

Буферы потоков

Потоки не читают и не записывают напрямую. Под капотом они используют классы потокового буфера. На высоком уровне *классы потокового буфера* — это шаблоны, которые отправляют или извлекают символы. Детали реализации не важны, если вы не планируете реализовать собственную потоковую библиотеку, но важно знать, что они существуют в нескольких контекстах. Потоковые буферы получаются с помощью метода `rdbuf` потока, который предоставляют все потоки.

Запись файлов в stdout

Иногда просто нужно записать содержимое потока входного файла непосредственно в поток вывода. Для этого можно извлечь указатель буфера потока из потока файлов и передать его оператору вывода. Например, можно вывести содержимое файла в стандартный вывод с помощью `cout` следующим образом:

```
cout << my_ifstream.rdbuf()
```

Да, вот так просто.

Итераторы буфера выходного потока

Итераторы буфера выходного потока — это шаблоны классов, которые предоставляют интерфейс итератора вывода, который преобразует записи в выходные операции в базовом буфере потока. Другими словами, это адаптеры, которые позволяют использовать выходные потоки, как если бы они были выходными итераторами.

Чтобы создать итератор буфера выходного потока, используйте класс шаблона `ostreambuf_iterator` в заголовке `<iterator>`. Его конструктор принимает один выходной аргумент потока и один параметр шаблона, соответствующий параметру шаблона аргумента конструктора (тип символа). В листинге 16.17 показано, как создать итератор буфера выходного потока из `cout`.

Листинг 16.17. Написание сообщения Hi для стандартного вывода с использованием класса ostreambuf_iterator

```
#include <iostream>
#include <iterator>

using namespace std;

int main() {
  ostreambuf_iterator<char> itr{ cout }; ❶
  *itr = 'H'; ❷
  ++itr; ❸
  *itr = 'i'; ❹
}
--------------------------------------------------------------------------------
H❷i❹
```

Здесь создается итератор буфера выходного потока из `cout` ❶, в который данные записываются обычным способом для оператора вывода: присвоить ❷, увеличить ❸, присвоить ❹ и т. д. Результатом является посимвольный вывод в стандартный вывод. (Вспомните процедуры обработки операторов вывода в разделе «Итераторы вывода» на с. 550.)

Итераторы буфера входного потока

Итераторы буфера входного потока — это шаблоны классов, которые предоставляют интерфейс итератора ввода, который преобразует операции чтения в операции

чтения в базовом буфере потока. Они полностью аналогичны итераторам буфера выходного потока.

Чтобы создать итератор буфера входного потока, используйте шаблонный класс `istreambuf_iterator` в заголовке `<iterator>`. В отличие от `ostreambuf_iterator`, он принимает аргумент буфера потока, поэтому нужно вызывать `rdbuf()` для любого входного потока, который планируется адаптировать. Этот аргумент является необязательным: конструктор по умолчанию `istreambuf_iterator` соответствует итератору конца диапазона входного итератора. Например, в листинге 16.18 показано, как создать строку из `std::cin`, используя основанный на диапазоне конструктор `string`.

Листинг 16.18. Построение string из cin с использованием итераторов буфера входного потока

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

using namespace std;

int main() {
  istreambuf_iterator<char> cin_itr{ cin.rdbuf() } ❶, end{} ❷;
  cout << "What is your name? "; ❸
  const string name{ cin_itr, end }; ❹
  cout << "\nGoodbye, " << name; ❺
}
--------------------------------------------------------------------------------
What is your name? ❸josh ❹
Goodbye, josh ❺
```

`istreambuf_iterator` создается из буфера потока `cin` ❶, а также итератора конца диапазона ❷. После отправки приглашения пользователю программы ❸ создается `string name` с использованием конструктора на основе диапазона ❹. Когда пользователь отправляет ввод (завершается EOF), конструктор строки копирует его. Затем вы прощаетесь с пользователем, используя `name` ❺. (Вспомните из раздела «Состояние потока» на с. 619, что способы отправки EOF на консоль отличаются в зависимости от операционной системы.)

Произвольный доступ

Иногда нужен произвольный доступ к потоку (особенно к потоку файлов). Операторы ввода и вывода явно не поддерживают этот вариант использования, поэтому `basic_istream` и `basic_ostream` предлагают отдельные методы для произвольного доступа. Эти методы отслеживают курсор или позицию, индекс текущего символа потока. Позиция указывает на следующий байт, который будет читать входной или выходной поток.

Для входных потоков можно использовать два метода — `tellg` и `seekg`. Метод `tellg` не принимает аргументов и возвращает позицию. Метод `seekg` позволяет установить

позицию курсора, и он имеет две перегрузки. Первый вариант — предоставить аргумент позиции `pos_type`, который устанавливает позицию чтения. Второй — предоставить аргумент смещения `off_type` плюс аргумент направления `ios_base::seekdir`. `Pos_type` и `off_type` определяются аргументами шаблона для `basic_istream` или `basic_ostream`, но обычно они преобразуются в/из целочисленных типов. Тип `seekdir` принимает одно из следующих трех значений:

- `ios_base::beg` указывает, что аргумент позиции располагается относительно начала;
- `ios_base::cur` указывает, что аргумент позиции располагается относительно текущей позиции;
- `ios_base::end` указывает, что аргумент позиции располагается относительно конца.

Для выходных потоков можно использовать два метода, `tellp` и `seekp`. Они примерно аналогичны методам `tellp` и `seekp` входных потоков: `p` обозначает `put`, а `g` обозначает `get`.

Рассмотрим файл *introspection.txt* со следующим содержимым:

```
The problem with introspection is that it has no end.
```

В листинге 16.19 показано, как использовать методы произвольного доступа для сброса курсора файла.

Листинг 16.19. Программа, использующая методы произвольного доступа для чтения произвольных символов в текстовом файле

```
#include <fstream>
#include <exception>
#include <iostream>

using namespace std;

ifstream open(const char* path, ios_base::openmode mode = ios_base::in) { ❶
--пропуск--
}

int main() {
  try {
    auto intro = open("introspection.txt"); ❷
    cout << "Contents: " << intro.rdbuf() << endl; ❸
    intro.seekg(0); ❹
    cout << "Contents after seekg(0): " << intro.rdbuf() << endl; ❺
    intro.seekg(-4, ios_base::end); ❻
    cout << "tellg() after seekg(-4, ios_base::end): "
                                                  << intro.tellg() << endl; ❼
    cout << "Contents after seekg(-4, ios_base::end): "
                                                  << intro.rdbuf() << endl; ❽
  }
```

```
  catch (const exception& e) {
    cerr << e.what();
  }
}
-----------------------------------------------------------------------------
Contents: The problem with introspection is that it has no end. ❸
Contents after seekg(0): The problem with introspection is that it has no end. ❺
tellg() after seekg(-4, ios_base::end): 49 ❼
Contents after seekg(-4, ios_base::end): end. ❽
```

С использованием фабричной функции в листинге 16.16 ❶ открывается текстовый файл *introspection.txt* ❷. Затем содержимое выводится в стандартный вывод с помощью метода `rdbuf` ❸, курсор перемещается на первый символ ❹ и снова выводится содержимое. Обратите внимание, что они дают идентичный вывод (поскольку файл не изменился) ❺. Затем используется относительная перегрузка смещения `seekg` для перехода к четвертому символу от конца ❻. Используя `tellg`, вы узнаете, что это 49-й символ (с индексированием на основе нуля) ❼. Когда входной файл выводится в стандартный вывод, результатом является только `end`, потому что это последние четыре символа в файле ❽.

ПРИМЕЧАНИЕ

Boost предлагает библиотеку IOStream с богатым набором дополнительных функций, которых нет в stdlib, в том числе средства ввода-вывода отображаемых в память файлов, сжатия и фильтрации.

Итоги

В этой главе вы узнали о потоках — основной концепции, которая обеспечивает общую абстракцию для выполнения операций ввода-вывода. Вы также узнали о файлах в качестве основного источника и назначения для ввода-вывода. Мы изучили фундаментальные классы потока в stdlib и то, как выполнять форматированные и неформатированные операции, проверять состояние потока и обрабатывать ошибки с исключениями. Поговорили о манипуляторах и о том, как включать потоки в определяемые пользователем типы, потоки строк и потоки файлов. Эта глава завершилась итераторами буфера потока, которые позволяют адаптировать поток к итератору.

Упражнения

16.1. Реализуйте выходной оператор, который выводит информацию об `AutoBrake` из раздела «Расширенный пример: `AutoBrake`» на с. 353. Добавьте текущий порог столкновения автомобиля и скорость.

16.2. Напишите программу, которая принимает результат из стандартного ввода, использует его заглавные буквы и записывает результат в стандартный вывод.

16.3. Прочитайте вводную документацию по Boost IOStream.

16.4. Напишите программу, которая принимает путь к файлу, открывает файл и печатает сводную информацию о содержимом, включая количество слов, среднюю длину слова и гистограмму символов.

Что еще почитать?

- «Standard C++ IOStreams and Locales: Advanced Programmer's Guide and Reference», Angelika Langer (Addison-Wesley Professional, 2000)
- Международный стандарт ИСО/МЭК (2017) — Язык программирования C++ (Международная организация по стандартизации; Женева, Швейцария; isocpp.org/std/the-standard/)
- «The Boost C++ Libraries», 2nd Edition, Boris Schäling (XML Press, 2014)

17 Файловые системы

«А вы, оказывается, у нас компьютерный гений». В ту пору Рэнди был глуп и наивен. Он клюнул на лесть, хотя должен был похолодеть от ужаса.

Нил Стивенсон, «Криптономикон»

В этой главе вы узнаете, как использовать библиотеку файловой системы stdlib для выполнения операций с файловыми системами, таких как управление файлами и проверка файлов, перечисление каталогов и взаимодействие с файловыми потоками.

stdlib и Boost содержат библиотеки файловой системы. Библиотека файловой системы stdlib выросла из Boost, и, соответственно, они в значительной степени взаимозаменяемы. Эта глава посвящена реализации stdlib. Если вы хотите узнать больше о Boost, обратитесь к документации Boost по файловой системе. Реализации Boost и stdlib в основном идентичны.

ПРИМЕЧАНИЕ

Стандарт C++ имеет историю использования библиотек Boost. Это позволяет сообществу C++ получить опыт работы с новыми функциями в Boost, прежде чем перейти к более сложному процессу включения функций в стандарт C++.

Концепты файловых систем

Файловые системы моделируют несколько важных концептов. Центральным объектом является файл. *Файл* — это объект файловой системы, который поддерживает

ввод и вывод и содержит данные. Файлы существуют в контейнерах, называемых *каталогами*, которые могут быть вложены в другие каталоги. Для простоты каталоги считаются файлами. Каталог, содержащий файл, называется *родительским каталогом* этого файла.

Путь — это строка, которая идентифицирует конкретный файл. Пути начинаются с необязательного *корневого имени*, которое представляет собой строку для конкретной реализации, например *C:* или *//localhost* в Windows, за которой следует необязательный корневой каталог, который представляет собой другую строку для конкретной реализации, такую как / в Unix-подобных системах. Остальная часть пути представляет собой последовательность каталогов, разделенных определенными реализацией разделителями. Необязательно, но пути могут заканчиваться файлом (не каталогом). Пути могут содержать специальные имена «.» и «..», что означает текущий каталог и родительский каталог соответственно.

Жесткая ссылка — это запись каталога, которая присваивает имя существующему файлу, а *символическая ссылка* присваивает имя пути (который может существовать или не существовать). Путь, местоположение которого указано относительно другого пути (обычно текущего каталога), называется *относительным путем* и *каноническим путем*, однозначно определяющим местоположение файла, не содержащим специальных имен «.» и «..» и не содержащим никаких символических ссылок. *Абсолютный путь* — это любой путь, который однозначно определяет местоположение файла. Основное различие между каноническим путем и абсолютным путем состоит в том, что канонический путь не может содержать специальных имен «.» и "..".

ПРЕДУПРЕЖДЕНИЕ

Файловая система `stdlib` может быть недоступна, если целевая платформа не предлагает иерархическую файловую систему.

std::filesystem::path

`std::filesystem::path` — это класс библиотеки Filesystem для моделирования пути, и есть много вариантов создания путей. Возможно, двумя наиболее распространенными являются конструктор по умолчанию, который создает пустой путь, и конструктор, принимающий строковый тип, который создает путь, указанный символами в строке. Как и все другие классы и функции файловой системы, класс `path` находится в заголовке `<filesystem>`.

В этом разделе вы узнаете, как построить путь из представления `string`, разложить его на составные части и изменить его. Во многих общих контекстах системного и прикладного программирования необходимо взаимодействовать с файлами. Поскольку каждая операционная система имеет уникальное представление для файловых систем, библиотека файловой системы stdlib является желанной абстракцией, которая позволяет легко писать кроссплатформенный код.

Создание path

Класс `path` поддерживает сравнение с другими объектами `path` и `string`, используя оператор `==`. Но если нужно просто проверить, является ли путь пустым, предлагается метод `empty`, который возвращает логическое значение. В листинге 17.1 показано, как построить два пути (один пустой и один непустой) и проверить их.

Листинг 17.1. Создание std::filesystem::path

```
#include <string>
#include <filesystem>

TEST_CASE("std::filesystem::path supports == and .empty()") {
  std::filesystem::path empty_path; ❶
  std::filesystem::path shadow_path{ "/etc/shadow" }; ❷
  REQUIRE(empty_path.empty()); ❸
  REQUIRE(shadow_path == std::string{ "/etc/shadow" }); ❹
}
```

В примере создаются два пути: один с конструктором по умолчанию ❶, а другой ссылается на `/etc/shadow` ❷. Поскольку `empty_path` создается по умолчанию, метод `empty` возвращает `true` ❸. `shadow_path` равен `string`, содержащей `/etc/shadow`, потому что она создается с тем же содержимым ❹.

Декомпозиция path

Класс `path` содержит некоторые методы декомпозиции, которые, по сути, являются специализированными строковыми манипуляторами, позволяющими извлекать компоненты пути, например:

- `root_name()` возвращает имя корня;
- `root_directory()` возвращает корневой каталог;
- `root_path()` возвращает корневой путь;
- `relative_path()` возвращает путь относительно корня;
- `parent_path()` возвращает родительский путь;
- `filename()` возвращает компонент имени файла;
- `stem()` возвращает имя файла без его расширения;
- `extension()` возвращает расширение.

В листинге 17.2 приведены значения, возвращаемые каждым из этих методов для пути, указывающего на очень важную системную библиотеку Windows, kernel32.dll.

Путь к `kernel32` создается с использованием обычного строкового литерала, чтобы экранировать обратную косую черту ❶. Извлекаются корневое имя ❷, корневой каталог ❸ и корневой путь kernel32 ❹ и выводятся в стандартный вывод. Затем извлекается относительный путь, который отображает путь относительно корня `C:\` ❺. Родительский путь — это путь родителя `kernel32.dll`, который является

просто каталогом, содержащим его ❻. Наконец, извлекается имя файла ❼, его основа ❽ и расширение ❾.

Листинг 17.2. Программа, выводящая различные примеры декомпозиции пути

```
#include <iostream>
#include <filesystem>

using namespace std;

int main() {
  const filesystem::path kernel32{ R"(C:\Windows\System32\kernel32.dll)" }; ❶
  cout << "Root name: " << kernel32.root_name() ❷
    << "\nRoot directory: " << kernel32.root_directory() ❸
    << "\nRoot path: " << kernel32.root_path() ❹
    << "\nRelative path: " << kernel32.relative_path() ❺
    << "\nParent path: " << kernel32.parent_path() ❻
    << "\nFilename: " << kernel32.filename() ❼
    << "\nStem: " << kernel32.stem() ❽
    << "\nExtension: " << kernel32.extension() ❾
    << endl;
}
--------------------------------------------------------------------------------
Root name: "C:" ❷
Root directory: "\\" ❸
Root path: "C:\\" ❹
Relative path: "Windows\\System32\\kernel32.dll" ❺
Parent path: "C:\\Windows\\System32" ❻
Filename: "kernel32.dll" ❼
Stem: "kernel32" ❽
Extension: ".dll" ❾
```

Обратите внимание, что листинг 17.2 не нужно запускать в какой-либо конкретной операционной системе. Ни один из методов декомпозиции не требует, чтобы путь фактически указывал на существующий файл. Здесь просто извлекаются компоненты содержимого пути, а не указанный файл. Конечно, разные операционные системы будут давать разные результаты, особенно в отношении разделителей (например, косой черты в Linux).

ПРИМЕЧАНИЕ

В листинге 17.2 показано, что в `std::filesystem::path` есть `operator<<`, который печатает кавычки в начале и в конце пути. Внутренне он использует `std::quoted`, шаблон класса в заголовке `<iomanip>`, который облегчает вставку и извлечение строк в кавычках. Кроме того, помните, что нужно избегать обратного слеша в строковом литерале, поэтому вы видите два, а не один в путях, встроенных в исходный код.

Изменение path

В дополнение к методам декомпозиции `path` предлагает несколько *методов-модификаторов*, которые позволяют изменять различные характеристики пути:

- `clear()` очищает путь;
- `make_preferred()` преобразует все разделители каталогов в предпочтительный для реализации разделитель каталогов. Например, в Windows он преобразует общий разделитель/в системный разделитель \;
- `remove_filename()` удаляет часть имени файла из пути;
- `replace_filename(p)` заменяет имя `path` на имя другого `path` `p`;
- `replace_extension(p)` заменяет расширение `path` на расширение другого пути `p`;
- `remove_extension()` удаляет часть расширения из пути.

В листинге 17.3 показано, как управлять путем, используя несколько методов-модификаторов.

Листинг 17.3. Управление путем с использованием методов-модификаторов. (Вывод из системы Windows 10 x64.)

```
#include <iostream>
#include <filesystem>

using namespace std;

int main() {
  filesystem::path path{ R"(C:/Windows/System32/kernel32.dll)" };
  cout << path << endl; ❶

  path.make_preferred();
  cout << path << endl; ❷

  path.replace_filename("win32kfull.sys");
  cout << path << endl; ❸

  path.remove_filename();
  cout << path << endl; ❹

  path.clear();
  cout << "Is empty: " << boolalpha << path.empty() << endl; ❺
}
--------------------------------------------------------------------------------
"C:/Windows/System32/kernel32.dll" ❶
"C:\\Windows\\System32\\kernel32.dll" ❷
"C:\\Windows\\System32\\win32kfull.sys" ❸
"C:\\Windows\\System32\\" ❹
Is empty: true ❺
```

Как и в листинге 17.2, создается путь к `kernel32`, хотя этот путь не является `const`, потому что мы собираемся его изменить ❶. Затем все разделители каталогов конвертируются в предпочтительный системный разделитель каталогов с помощью `make_preferred`. В листинге 17.3 показан вывод системы Windows 10 x64, поэтому в ней слеш (`/`) преобразован в обратный слеш (`\`) ❷. Используя `replace_filename`, имя файла заменяется с `kernel32.dll` на `win32kfull.sys` ❸. Еще раз обратите вни-

мание, что файл, описанный по этому пути, не обязательно должен существовать в системе; вы просто манипулируете путем. Наконец, имя файла удаляется с помощью метода `remove_filename` ❹, а затем содержимое пути полностью очищается, используя `clear` ❺.

Обзор методов path файловой системы

Таблица 17.1 содержит неполный список доступных методов `path`. Обратите внимание, что `p`, `p1` и `p2` означают объекты `path`, а `s` — `stream` в таблице.

Таблица 17.1. Сводка операций `std::filestystem::path`

Операция	Примечания
`path{}`	Создает пустой путь
`Path{ s, [f] }`	Создает путь из строки типа `s`; `f` — это необязательный тип `path::format`, который по умолчанию соответствует формату пути, определяемому реализацией
`Path{ p }` `p1 = p2`	Конструктор/присваивание копирования
`Path{ move(p) }` `p1 = move(p2)`	Конструктор/присваивание перемещения
`p.assign(s)`	Присваивает `p` `s`, отбрасывая текущее содержимое
`p.append(s)` `p / s`	Добавляет `s` к `p`, включая соответствующий разделитель, `path::preferred_separator`
`p.concat(s)` `p + s`	Добавляет `s` к `p` без добавления разделителя
`p.clear()`	Стирает содержимое
`p.empty()`	Возвращает `true`, если `p` пуст
`p.make_preferred()`	Преобразует все разделители каталогов в предпочтительный для реализации разделитель каталогов
`p.remove_filename()`	Удаляет часть с именем файла
`p1.replace_filename(p2)`	Заменяет имя файла `p1` именем файла `p2`
`p1.replace_extension(p2)`	Заменяет расширение `p1` расширением `p2`
`p.root_name()`	Возвращает корневое имя
`p.root_directory()`	Возвращает корневой каталог
`p.root_path()`	Возвращает корневой путь
`p.relative_path()`	Возвращает относительный путь

Продолжение ⇩

Таблица 17.1 *(продолжение)*

Операция	Примечания
`p.parent_path()`	Возвращает родительский путь
`p.filename()`	Возвращает имя файла
`p.stem()`	Возвращает основу
`p.extension()`	Возвращает расширение
`p.has_root_name()`	Возвращает `true`, если `p` включает корневое имя
`p.has_root_directory()`	Возвращает `true`, если `p` включает корневой каталог
`p.has_root_path()`	Возвращает `true`, если `p` включает корневой путь
`p.has_relative_path()`	Возвращает `true`, если `p` включает относительный путь
`p.has_parent_path()`	Возвращает `true`, если `p` включает родительский путь
`p.has_filename()`	Возвращает `true`, если `p` включает имя файла
`p.has_stem()`	Возвращает `true`, если `p` включает основу
`p.has_extension()`	Возвращает `true`, если `p` включает расширение
`p.c_str()` `p.native()`	Возвращает встроенное строковое представление `p`
`p.begin()` `p.end()`	Получает доступ к элементам пути последовательно в виде полуоткрытого диапазона
`s << p`	Записывает `p` в `s`
`s >> p`	Читает `s` в `p`
`p1.swap(p2)` `swap(p1, p2)`	Меняет все элементы `p1` на элементы `p2`

`p1 == p2`
`p1 != p2`
`p1 > p2`
`p1 >= p2`
`p1 < p2`
`p1 <= p2`
	Лексикографически сравнивает два пути — `p1` и `p2`

Файлы и каталоги

Класс `path` является центральным элементом библиотеки файловой системы, но ни один из его методов на самом деле не взаимодействует с файловой системой. Заголовок `<filesystem>` содержит для этого функции, не являющиеся членами. Представьте объекты `path` как способ объявления, с какими компонентами файловой системы вы хотите взаимодействовать, и представьте заголовок `<filesystem>` как содержащий функции, которые выполняют работу с этими компонентами.

Эти функции имеют дружественные интерфейсы обработки ошибок и позволяют разбивать пути, например на имя каталога, имя файла и расширение. Используя эти функции, вы получаете много инструментов для взаимодействия с файлами в среде без необходимости использования интерфейса прикладного программирования, специфичного для конкретной операции.

Обработка ошибок

Взаимодействие с файловой системой среды может привести к ошибкам вроде ненайденных файлов, недостаточных разрешений или неподдерживаемых операций. Поэтому каждая функция, не являющаяся членом в библиотеке файловой системы, но взаимодействующая с файловой системой, должна сообщать вызывающей стороне об ошибках. Эти функции, не являющиеся членами, предоставляют две опции: генерировать исключение или устанавливать переменную ошибки.

Каждая функция имеет две перегрузки: одна позволяет передавать ссылку на `std::system_error`, а другая — пропустить этот параметр. Если предоставить ссылку, функция установит `system_error` равным условию ошибки, если та возникнет. Если не предоставить эту ссылку, функция вместо этого выдаст `std::filesystem::filesystem_error` (тип исключения, унаследованный от `std::system_error`).

Функции композиции path

В качестве альтернативы использованию конструктора `path` можно создавать различные виды путей:

- `absolute(p, [ec])` возвращает абсолютный путь, ссылающийся на то же местоположение, что и **`p`**, но где `is_absolute()` равен `true`;
- `canonical(p, [ec])` возвращает канонический путь, ссылающийся на то же местоположение, что и **`p`**;
- `current_path([ec])` возвращает текущий путь;
- `relative(p, [base], [ec])` возвращает путь, где **`p`** сделан относительным к **`base`**;
- `temp_directory_path([ec])` возвращает каталог для временных файлов. Результатом гарантированно будет существующий каталог.

Обратите внимание, что `current_path` поддерживает перегрузку, поэтому можно установить текущий каталог (как в **`cd`** или **`chdir`** в Posix). Просто укажите аргумент пути, как в `current_path (p, [ec])`.

В листинге 17.4 показана работа некоторых из этих функций.

Путь создается, используя `temp_directory_path`, который возвращает системный каталог для временных файлов ❶, а затем используется, чтобы определить его относительный путь ❷. После вывода временного пути ❹ `is_absolute` показывает, что этот путь является абсолютным ❹. Затем выводится текущий путь ❺ и путь временного каталога относительно текущего пути ❻. Поскольку этот путь относи-

телен, `is_absolute` возвращает `false` ❼. Как только путь изменится на временный ❽, выведется текущий каталог ❾. Конечно, вывод будет отличаться от вывода в листинге 17.4, и вы можете даже получить исключение, если ваша система не поддерживает определенные операции ❿. (Вспомните предупреждение в начале главы: стандарт C++ допускает, что некоторые среды могут не поддерживать определенные или все библиотеки файловой системы.)

Листинг 17.4. Программа, использующая несколько функций создания пути. (Вывод из системы Windows 10 x64.)

```
#include <filesystem>
#include <iostream>

using namespace std;

int main() {
  try {
    const auto temp_path = filesystem::temp_directory_path();❶
    const auto relative = filesystem::relative(temp_path); ❷
    cout << boolalpha
      << "Temporary directory path: " << temp_path ❸
      << "\nTemporary directory absolute: " << temp_path.is_absolute() ❹
      << "\nCurrent path: " << filesystem::current_path() ❺
      << "\nTemporary directory's relative path: " << relative ❻
      << "\nRelative directory absolute: " << relative.is_absolute() ❼
      << "\nChanging current directory to temp.";

    filesystem::current_path(temp_path); ❽
    cout << "\nCurrent directory: " << filesystem::current_path(); ❾
  } catch(const exception& e) {❿
    cerr << "Error: " << e.what();
  }
}
--------------------------------------------------------------------------------
Temporary directory path: "C:\\Users\\lospi\\AppData\\Local\\Temp\\"❸
Temporary directory absolute: true ❹
Current path: "c:\\Users\\lospi\\Desktop" ❺
Temporary directory's relative path: "..\\AppData\\Local\\Temp" ❻
Relative directory absolute: false ❼
Changing current directory to temp. ❽
Current directory: "C:\\Users\\lospi\\AppData\\Local\\Temp" ❾
```

Просмотр типов файлов

Можно проверить атрибуты файла по заданному пути, используя следующие функции:

- `is_block_file(p, [ec])` определяет, является ли `p` *блочным файлом*, специальным файлом в некоторых операционных системах (например, в блочных устройствах в Linux, которые позволяют передавать произвольно доступные данные в блоки фиксированного размера);

- `is_character_file(p, [ec])` определяет, является ли `p` *символьным файлом*, специальным файлом в некоторых операционных системах (например, символьные устройства в Linux, которые позволяют отправлять и получать отдельные символы);
- `is_regular_file(p, [ec])` определяет, является ли `p` обычным файлом;
- `is_symlink(p, [ec])` определяет, является ли `p` символической ссылкой, которая является ссылкой на другой файл или каталог;
- `is_empty(p, [ec])` определяет, является ли `p` пустым файлом или каталогом;
- `is_directory(p, [ec])` определяет, является ли `p` каталогом;
- `is_fifo(p, [ec])` определяет, является ли `p` *именованным каналом*, особым видом механизма межпроцессного взаимодействия во многих операционных системах;
- `is_socket(p, [ec])` определяет, является ли `p` *сокетом*, другим специальным механизмом межпроцессного взаимодействия во многих операционных системах;
- `is_other (p, [ec])` определяет, является ли `p` каким-либо файлом, отличным от обычного файла, каталога или символической ссылки.

Листинг 17.5 использует `is_directory` и `is_regular_file` для проверки четырех разных путей.

Листинг 17.5. Программа, которая проверяет традиционные пути Windows и Linux с помощью is_directory и is_regular_file

```
#include <iostream>
#include <filesystem>

using namespace std;

void describe(const filesystem::path& p) { ❶
  cout << boolalpha << "Path: " << p << endl;
  try {
    cout << "Is directory: " << filesystem::is_directory(p) << endl; ❷
    cout << "Is regular file: " << filesystem::is_regular_file(p) << endl; ❸
  } catch (const exception& e) {
    cerr << "Exception: " << e.what() << endl;
  }
}

int main() {
  filesystem::path win_path{ R"(C:/Windows/System32/kernel32.dll)" };
  describe(win_path); ❹
  win_path.remove_filename();
  describe(win_path); ❺

  filesystem::path nix_path{ R"(/bin/bash)" };
  describe(nix_path); ❻
  nix_path.remove_filename();
  describe(nix_path); ❼
}
```

На компьютере под управлением Windows 10 x64 запуск программы из листинга 17.5 дал следующий результат:

```
Path: "C:/Windows/System32/kernel32.dll" ❹
Is directory: false ❹
Is regular file: true ❹
Path: "C:/Windows/System32/" ❺
Is directory: true ❺
Is regular file: false ❺
Path: "/bin/bash" ❻
Is directory: false ❻
Is regular file: false ❻
Path: "/bin/" ❼
Is directory: false ❼
Is regular file: false ❼
```

И на машине с Ubuntu 18.04 x64 запуск программы из листинга 17.5 привел к следующему выводу:

```
Path: "C:/Windows/System32/kernel32.dll" ❹
Is directory: false ❹
Is regular file: false ❹
Path: "C:/Windows/System32/" ❺
Is directory: false ❺
Is regular file: false ❺
Path: "/bin/bash" ❻
Is directory: false ❻
Is regular file: true ❻
Path: "/bin/" ❼
Is directory: true ❼
Is regular file: false ❼
```

Сначала определяется функция `describe`, которая принимает единственный `path` ❶. После вывода пути также выводится, является ли путь каталогом ❷ или обычным файлом ❸. В рамках `main` передаются различные пути для описания:

- C:/Windows/System32/kernel32.dll ❹
- C:/Windows/System32/ ❺
- /bin/bash ❻
- /bin/ ❼

Обратите внимание, что результат зависит от операционной системы.

Просмотр файлов и каталогов

Можно проверять различные атрибуты файловой системы, используя следующие функции:

- `current_path([p], [ec])`, которая, если указан **`p`**, устанавливает текущий путь программы как `p`; в противном случае возвращает текущий путь программы;
- `exist(p, [ec])`проверяет, существует ли файл или каталог в **`p`**;

- `equivalent(p1, p2, [ec])`проверяет, ссылаются ли **`p1`** и **`p2`** на один и тот же файл или каталог;
- `file_size(p, [ec])` возвращает размер в байтах обычного файла в **`p`**;
- `hard_link_count(p, [ec])` возвращает количество жестких ссылок для **`p`**;
- `last_write_time(p, [t] [ec])`, которая, если указан **`t`**, устанавливает время последнего изменения **`p`** в **`t`**; в противном случае возвращается последний раз, когда `p` был изменен (`t` является `std::chrono::time_point`);
- `permissions(p, prm, [ec])` устанавливают права доступа **`p`**. **`prm`** имеет тип `std::filesystem::perms`, который является классом `enum`, смоделированным после битов разрешения POSIX (см. [fs.enum.perms]);
- `read_symlink(p, [ec])` возвращает цель символической ссылки **`p`**;
- `space(p, [ec])` возвращает пространственную информацию о файловой системе **`p`**, занимаемой в виде `std::filesystem::space_info`. Этот POD содержит три поля: `capacity` (общий размер), `free` (свободное пространство) и `available` (свободное пространство, доступное для непривилегированного процесса). Все они представляют собой целочисленный тип без знака, измеряемый в байтах;
- `status(p, [ec])` возвращает тип и атрибуты файла или каталога **`p`** в виде `std::filesystem::file_status`. Этот класс содержит метод типа, который не принимает параметров и возвращает объект типа `std::filesystem::file_type`, который является классом `enum`, принимающим значения, описывающие тип файла, например `not_found`, `normal`, `directory`. Класс `symlinkfile_status` также предлагает метод `permissions`, который не принимает параметров и возвращает объект типа `std::filesystem::perms` (подробнее см. [fs.class.file_status]);
- `symlink_status(p, [ec])` похож на `status`, который не будет следовать символическим ссылкам.

Если вы знакомы с Unix-подобными операционными системами, то, несомненно, много раз использовали программу `ls` (сокращение от «list») для просмотра файлов и каталогов. В DOS-подобных операционных системах (включая Windows) есть аналогичная команда `dir`. Вы будете использовать некоторые из этих функций позже в этой главе (в листинге 17.7), чтобы создать свою собственную простую программу просмотра файлов и каталогов.

Теперь, когда вы знаете, как проверять файлы и каталоги, давайте перейдем к тому, как можно манипулировать файлами и каталогами, на которые ссылаются пути.

Управление файлами и каталогами

Кроме того, библиотека файловой системы содержит ряд методов для управления файлами и каталогами:

- `copy(p1, p2, [opt],[ec])` копирует файлы или каталоги из **`p1`** в **`p2`**. Можно предоставить опцию `std::filesystem::copy_options` для настройки поведения `copy_file`.

Этот класс `enum` может принимать несколько значений, в том числе ни одного (сообщить об ошибке, если место назначения уже существует), `skip_existing` (сохранить существующее), `overwrite_existing` (перезаписать) и `update_existing` (перезаписать, если p1 новее) (подробнее см. [fs.enum.copy.opts]);

- `copy_file(p1, p2, [opt], [ec])` похож на `copy`, за исключением того, что он выдаст ошибку, если **`p1`** — это не обычный файл;
- `create_directory(p, [ec])` создает каталог **`p`**;
- `create_directories(p, [ec])` похож на рекурсивный вызов `create_directory`, поэтому, если вложенный путь содержит несуществующих родителей, используйте эту форму;
- `create_hard_link(tgt, lnk, [ec])` создает жесткую ссылку на **`tgt`** в **`lnk`**;
- `create_symlink(tgt, lnk, [ec])` создает символическую ссылку на **`tgt`** в **`lnk`**;
- `create_directory_symlink(tgt, lnk, [ec])` следует использовать для каталогов вместо `create_symlink`;
- `remove(p, [ec])` удаляет файл или пустой каталог **`p`** (без следования символическим ссылкам);
- `remove_all(p, [ec])` рекурсивно удаляет файл или каталог **`p`** (без следования символическим ссылкам);
- `rename(p1, p2, [ec])` переименовывает **`p1`** в **`p2`**;
- `resize_file(p, new_size, [ec])` изменяет размер **`p`** (если это обычный файл) на **`new_size`**. Если эта операция увеличивает файл, новое пространство заполняется нулями. В противном случае операция обрезает **`p`** с конца.

Можно создать программу, которая копирует, изменяет размеры и удаляет файл, используя некоторые из этих методов. Листинг 17.6 показывает это, определяя функцию, которая выводит размер файла и время модификации. В `main` программа создает и изменяет два объекта пути и вызывает эту функцию после каждой модификации.

Листинг 17.6. Программа, где показаны несколько методов взаимодействия с файловой системой. (Вывод из системы Windows 10 x64.)

```
#include <iostream>
#include <filesystem>

using namespace std;
using namespace std::filesystem;
using namespace std::chrono;

void write_info(const path& p) {
  if (!exists(p)) { ❶
    cout << p << " does not exist." << endl;
    return;
  }
  const auto last_write = last_write_time(p).time_since_epoch();
  const auto in_hours = duration_cast<hours>(last_write).count();
```

```
  cout << p << "\t" << in_hours << "\t" << file_size(p) << "\n"; ❷
}

int main() {
  const path win_path{ R"(C:/Windows/System32/kernel32.dll)" }; ❸
  const auto reamde_path = temp_directory_path() / "REAMDE"; ❹
  try {
    write_info(win_path); ❺
    write_info(reamde_path); ❻

    cout << "Copying " << win_path.filename()
         << " to " << reamde_path.filename() << "\n";
    copy_file(win_path, reamde_path);
    write_info(reamde_path); ❼

    cout << "Resizing " << reamde_path.filename() << "\n";
    resize_file(reamde_path, 1024);
    write_info(reamde_path); ❽

    cout << "Removing " << reamde_path.filename() << "\n";
    remove(reamde_path);
    write_info(reamde_path); ❾
  } catch(const exception& e) {
    cerr << "Exception: " << e.what() << endl;
  }
}
-------------------------------------------------------------------------------
"C:/Windows/System32/kernel32.dll"      3657767 720632 ❺
"C:\\Users\\lospi\\AppData\\Local\\Temp\\REAMDE" does not exist. ❻
Copying "kernel32.dll" to "REAMDE"
"C:\\Users\\lospi\\AppData\\Local\\Temp\\REAMDE"        3657767 720632 ❼
Resizing "REAMDE"
"C:\\Users\\lospi\\AppData\\Local\\Temp\\REAMDE"        3659294 1024 ❽
Removing "REAMDE"
"C:\\Users\\lospi\\AppData\\Local\\Temp\\REAMDE" does not exist. ❾
```

Функция `write_info` принимает один параметр пути. Код проверяет, существует ли этот путь ❶, выводя сообщение об ошибке и немедленно завершая работу, если его нет. Если путь существует, выводится сообщение с указанием времени последнего изменения (в часах с начала эпохи) и размера файла ❷.

В `main` создается путь `win_path` к `kernel32.dll` ❸ и путь к несуществующему файлу `REAMDE` во временном каталоге файловой системы по адресу `reamde_path` ❹. (Вспомните из таблицы 17.1, что можно использовать `operator/` для объединения двух объектов пути.) Внутри блока `try-catch` вызывается информацию `write_` для обоих путей ❺ ❻. (Если используется машина не под управлением Windows, будет другой вывод. Можно изменить `win_path` на существующий файл в системе, чтобы продолжить.)

Затем файл копируется из `win_path` в `reamde_path` и для него вызывается `write_info` ❼. Обратите внимание, что в отличие от более раннего ❻, файл в `reamde_path` существует и имеет то же время последней записи и размер файла, что и `kernel32.dll`.

Затем размер файла в `reamde_path` изменяется до 1024 байтов и вызывается `write_info` ❽. Обратите внимание, что время последней записи увеличилось с 3657767 до 3659294, а размер файла уменьшился с 720632 до 1024.

Наконец, файл в `reamde_path` удаляется и вызывается `write_info` ❾, который сообщает, что файл больше не существует.

ПРИМЕЧАНИЕ

То, как файловые системы изменяют размер файла под капотом, зависит от операционной системы и выходит за рамки этой книги. Но в целом операция изменения размера концептуально может работать как операция `resize` в `std::vector`. Все данные в конце файла, которые не вписываются в новый размер файла, удаляются операционной системой.

Итераторы каталогов

Библиотека файловой системы предоставляет два класса для перебора элементов каталога: `std::filesystem::directory_iterator` и `std::filesystem::recursive_directory_iterator`. `directory_iterator` не будет вводить подкаталоги, а `recursive_directory_iterator` будет. В этом разделе представлен только `directory_iterator`, но `recursive_directory_iterator` может быть заменой для него и поддерживает все следующие операции.

Создание

Конструктор по умолчанию для `directory_iterator` создает конечный итератор. (Вспомните, что итератор конца ввода указывает, когда диапазон ввода исчерпан.) Другой конструктор принимает путь, указывающий каталог, который нужно перебрать. При желании можно предоставить `std::filesystem::directory_options`, которая является битовой маской класса `enum` со следующими постоянными:

- `none` указывает, что итератор пропускает символьные ссылки на каталог. Если итератор встречает отказ в разрешении, он выдает ошибку;
- `follow_directory_symlink` следует за символическими ссылками;
- `skip_permission_denied` пропускает каталоги, если итератор встречает отказ в разрешении.

Кроме того, можно предоставить `std::error_code`, его, как и все остальные функции библиотеки файловой системы, которые принимают код ошибки, будет устанавливать этот параметр, а не выбрасывать исключение, если во время построения возникает ошибка.

Таблица 17.2 обобщает эти опции для создания `directory_iterator`.

Обратите внимание, что `p` означает `path`, а `d` — `directory`, `op` — `directory_options`, а `ec` — `error_code` в таблице.

Таблица 17.2. Сводка операций `std::filestystem::directory_iterator`

Операция	Примечания
`directory_iterator{}`	Создает конечный итератор
`directory_iterator{ p, [op], [ec] }`	Создает итератор каталога, ссылаясь на каталог `p`. Аргумент `op` по умолчанию имеет значение `none`. Если указано, `ec` получает условия ошибки, а не выдает исключение
`directory_iterator { d }` `d1 = d2`	Копирует создание/присваивание
`directory_iterator { move(d) }` `d1 = move(d2)`	Перемещает создание/присваивание

Записи каталогов

Итераторы ввода `directory_iterator` и `recursive_directory_iterator` создают элемент `std::filesystem::directory_entry` для каждой записи, с которой они сталкиваются. Класс `directory_entry` хранит путь, а также некоторые атрибуты этого пути, представленные в виде методов. В таблице 17.3 перечислены эти методы. Обратите внимание, что `de` означает `directory_entry` в таблице.

Таблица 17.3. Сводка операций `std::filesystem::directory_entry`

Операция	Описание
`de.path()`	Возвращает указанный путь
`de.exists()`	Возвращает `true`, если указанный путь существует в файловой системе
`de.is_block_file()`	Возвращает `true`, если указанный путь — путь к блочному устройству
`de.is_character_file()`	Возвращает `true`, если указанный путь — путь к символьному устройству
`de.is_directory()`	Возвращает `true`, если указанный путь — путь к каталогу
`de.is_fifo()`	Возвращает `true`, если указанный путь — путь к именованному каналу
`de.is_regular_file()`	Возвращает `true`, если указанный путь — путь к обычному файлу
`de.is_socket()`	Возвращает `true`, если указанный путь — путь к сокету
`de.is_symlink()`	Возвращает `true`, если указанный путь — путь к символической ссылке

Продолжение ⇩

Таблица 17.3 *(продолжение)*

Операция	Описание
`de.is_other()`	Возвращает `true`, если указанный путь — путь к чему-либо еще
`de.file_size()`	Возвращает размер указанного пути
`de.hard_link_count()`	Возвращает количество жестких ссылок на указанный путь
`de.last_write_time([t])`	Если задан `t`, задает время последнего изменения указанного пути; в противном случае возвращает время последнего изменения
`de.status()` `de.symlink_status()`	Возвращает `std::filesystem::file_status` для указанного пути

Можно использовать `directory_iterator` и несколько операций в таблице 17.3 для создания простой программы просмотра каталогов, как показано в листинге 17.7.

Листинг 17.7. Программа просмотра файлов и каталогов, использующая std::filesystem::directory _iterator для перечисления заданного каталога. (Вывод из системы Windows 10 x64.)

```
#include <iostream>
#include <filesystem>
#include <iomanip>

using namespace std;
using namespace std::filesystem;
using namespace std::chrono;

void describe(const directory_entry& entry) { ❶
  try {
    if (entry.is_directory()) { ❷
      cout << " *";
    } else {
      cout << setw(12) << entry.file_size();
    }
    const auto lw_time =
      duration_cast<seconds>(entry.last_write_time().time_since_epoch());
    cout << setw(12) << lw_time.count()
      << " " << entry.path().filename().string()
      << "\n"; ❸
  } catch (const exception& e) {
    cout << "Error accessing " << entry.path().string()
         << ": " << e.what() << endl; ❹
  }
}


int main(int argc, const char** argv) {
  if (argc != 2) {
    cerr << "Usage: listdir PATH";
```

```
    return -1; ❺
  }
  const path sys_path{ argv[1] }; ❻
  cout << "Size Last Write Name\n";
  cout << "------------ ----------- ------------\n"; ❼
  for (const auto& entry : directory_iterator{ sys_path }) ❽
    describe(entry); ❾
}
-------------------------------------------------------------------------------
> listdir c:\Windows
Size         Last Write Name
------------ ----------- ------------
           * 13177963504 addins
           * 13171360979 appcompat
--пропуск--
           * 13173551028 WinSxS
      316640 13167963236 WMSysPr9.prx
       11264 13167963259 write.exe
```

ПРИМЕЧАНИЕ

Следует изменить имя программы с `listdir` на любое значение, совпадающее с выводом компилятора.

Сначала определяется функция `describe`, которая принимает ссылку на путь ❶, проверяет, является ли путь каталогом ❷, и выводит звездочку для каталога и соответствующий размер для файла. Затем определяется последняя модификация записи в секундах с начала эпохи и выводится ее значение вместе со связанным с ней именем файла ❸. Если возникает какое-либо исключение, выводится сообщение об ошибке, и программа завершает работу ❹.

В `main` сначала осуществляется проверка, что пользователь вызывал программу с одним аргументом, и возвращается отрицательное число, если это не так ❺. Затем создается путь, используя единственный аргумент ❻, выводится несколько необычных заголовков ❼, каждый `entry` в каталоге перебирается ❽ и передается в `describe` ❾.

Рекурсивный перебор каталога

`recursive_directory_iterator` является заменой `directory_iterator` в том смысле, что он поддерживает все те же операции, но будет перечислять подкаталоги. Можно использовать эти итераторы в комбинации для создания программы, которая вычисляет размер и количество файлов и подкаталогов для данного каталога. Листинг 17.8 показывает, как это сделать.

ПРИМЕЧАНИЕ

Следует изменить имя программы с `treedir` на любое значение, совпадающее с выводом вашего компилятора.

Листинг 17.8. Программа просмотра файлов и каталогов, использующая std::filesystem::recursive _directory_iterator для отображения количества файлов и общего размера подкаталога этого пути. (Вывод из системы Windows 10 x64.)

```
#include <iostream>
#include <filesystem>

using namespace std;
using namespace std::filesystem;

struct Attributes {
  Attributes& operator+=(const Attributes& other) {
    this->size_bytes += other.size_bytes;
    this->n_directories += other.n_directories;
    this->n_files += other.n_files;
    return *this;
  }
  size_t size_bytes;
  size_t n_directories;
  size_t n_files;
}; ❶

void print_line(const Attributes& attributes, string_view path) {
  cout << setw(14) << attributes.size_bytes
       << setw(7) << attributes.n_files
       << setw(7) << attributes.n_directories
       << " " << path << "\n"; ❷
}

Attributes explore(const directory_entry& directory) {
  Attributes attributes{};
  for(const auto& entry : recursive_directory_iterator{ directory.path() }) { ❸
      if (entry.is_directory()) {
        attributes.n_directories++; ❹
      } else {
        attributes.n_files++;
        attributes.size_bytes += entry.file_size(); ❺
      }
  }
  return attributes;
}

int main(int argc, const char** argv) {
  if (argc != 2) {
    cerr << "Usage: treedir PATH";
    return -1; ❻
  }
  const path sys_path{ argv[1] };
  cout << "Size Files Dirs Name\n";
  cout << "-------------- ------ ------ ------------\n";
  Attributes root_attributes{};
  for (const auto& entry : directory_iterator{ sys_path }) { ❼
    try {
```

```
      if (entry.is_directory()) {
        const auto attributes = explore(entry); ❽
        root_attributes += attributes;
        print_line(attributes, entry.path().string());
        root_attributes.n_directories++;
      } else {
        root_attributes.n_files++;
        error_code ec;
        root_attributes.size_bytes += entry.file_size(ec); ❾
        if (ec) cerr << "Error reading file size: "
                     << entry.path().string() << endl;
      }
    } catch(const exception&) {
    }
  }
  print_line(root_attributes, argv[1]); ❿
}
-------------------------------------------------------------------------------
> treedir C:\Windows
Size         Files Dirs Name
------------ ----- ----- ------------
           802 1 0 C:\Windows\addins
       8267330 9 5 C:\Windows\apppatch
--пропуск--
   11396916465 73383 20480 C:\Windows\WinSxS
   21038460348 110950 26513 C:\Windows ❿
```

После объявления класса `Attributes` для хранения учетных данных ❶ определяется функция `print_line`, которая предоставляет экземпляр `Attributes` удобным для пользователя способом вместе со строкой пути ❷. Затем определяется функция `explore`, которая принимает ссылку на `directory_entry` и выполняет рекурсивное повторение ❸. Если полученная запись является каталогом, количество каталогов увеличивается ❹; в противном случае увеличиваются количество и общий размер файлов ❺.

Внутри `main` проверяется, что программа вызывается ровно с двумя аргументами. Если нет, возвращается код ошибки -1 ❻. Используется (нерекурсивный) `directory_iterator` для перечисления содержимого целевого пути, на который указывает `sys_path` ❼. Если запись является каталогом, вызывается `explore` для определения ее атрибутов ❽, которые впоследствии выводятся в консоль. Также увеличивается член `n_directories` в `root_attributes`, чтобы сохранить учетную запись. Если запись не является каталогом, соответственно увеличиваются `root_attributes n_files` и `size_bytes` ❾.

После того как итерация по всем подэлементам `sys_path` завершается, `root_attributes` выводится в качестве последней строки ❿. Например, последняя строка вывода в листинге 17.8 показывает, что этот конкретный каталог Windows содержит 110 950 файлов, занимающих 21 038 460 348 байт (около 21 ГБ) и 26 513 подкаталогов.

Взаимосовместимость в fstream

Можно создавать файловые потоки (`basic_ifstream`, `basic_ofstream` или `basic_fstream`), используя `std::filesystem::path` или `std::filesystem::directory_entry` в дополнение к строковым типам.

Например, можно перебирать каталог и создавать `ifstream` для чтения каждого встреченного файла. В листинге 17.9 показано, как проверять магические байты MZ в начале каждого переносимого исполняемого файла Windows (*.sys*, *.dll*, *.exe* и т. д.) и сообщать о любом файле, нарушающем это правило.

Листинг 17.9. Поиск в каталоге System32 операционной системы Windows переносимых исполняемых файлов Windows

```
#include <iostream>
#include <fstream>
#include <filesystem>
#include <unordered_set>

using namespace std;
using namespace std::filesystem;

int main(int argc, const char** argv) {
  if (argc != 2) {
    cerr << "Usage: pecheck PATH";
    return -1; ❶
  }
  const unordered_set<string> pe_extensions{
    ".acm", ".ax", ".cpl", ".dll", ".drv",
    ".efi", ".exe", ".mui", ".ocx", ".scr",
    ".sys", ".tsp"
  }; ❷
  const path sys_path{ argv[1] };
  cout << "Searching " << sys_path << " recursively.\n";
  size_t n_searched{};
  auto iterator = recursive_directory_iterator{ sys_path,
                                directory_options::skip_permission_denied }; ❸
    for (const auto& entry : iterator) { ❹
      try {
      if (!entry.is_regular_file()) continue;
      const auto& extension = entry.path().extension().string();
      const auto is_pe = pe_extensions.find(extension) != pe_extensions.end();
      if (!is_pe) continue; ❺
      ifstream file{ entry.path() }; ❻
      char first{}, second{};
      if (file) file >> first;
      if (file) file >> second; ❼
      if (first != 'M' || second != 'Z')
        cout << "Invalid PE found: " << entry.path().string() << "\n"; ❽
      ++n_searched;
    } catch(const exception& e) {
```

```
      cerr << "Error reading " << entry.path().string()
           << ": " << e.what() << endl;
    }
  }
  cout << "Searched " << n_searched << " PEs for magic bytes." << endl; ❾
}
--------------------------------------------------------------------------------
listing_17_9.exe c:\Windows\System32
Searching "c:\\Windows\\System32" recursively.
Searched 8231 PEs for magic bytes.
```

В `main` проверяются ровно два аргумента и возвращается код ошибки в зависимости от ситуации ❶. Создается `unordered_set`, содержащий все расширения, связанные с переносимыми исполняемыми файлами ❷, которые будут использованы для проверки расширений файлов. `recursive_directory_iterator` используется с опцией `directory_options::skip_permission_denied` для перечисления всех файлов по указанному пути ❸. Каждая запись перебирается ❹, пропуская все, что не является обычным файлом, и определяется, является ли запись переносимым исполняемым файлом, запуская `find` в `pe_extensions`. Если запись не имеет такого расширения, файл пропускается ❺.

Чтобы открыть файл, путь записи просто передается в конструктор `ifstream` ❻. Затем используется результирующий поток входного файла, чтобы прочитать первые два байта файла в `first` и `second` ❼. Если эти первые два символа не `MZ`, выводится сообщение в консоль ❽. В любом случае увеличивается счетчик с именем `n_searched`. После исчерпания итератора каталога выводится сообщение с указанием `n_searched` для пользователя, прежде чем вернуть результат из `main` ❾.

Итоги

В этой главе вы узнали о возможностях файловой системы stdlib, включая пути, файлы, каталоги и обработку ошибок. Эти средства позволяют писать кроссплатформенный код, который взаимодействует с файлами в вашей среде. Кульминацией главы стали некоторые важные операции, итераторы каталогов и взаимодействие с файловыми потоками.

Упражнения

17.1. Реализуйте программу, которая принимает два аргумента: путь и расширение. Программа должна рекурсивно искать указанный путь и выводить любой файл с указанным расширением.

17.2. Улучшите программу в листинге 17.8, чтобы она могла принимать необязательный второй аргумент. Если первый аргумент начинается с дефиса (-), программа считывает все смежные буквы сразу после

дефиса и анализирует каждую букву в качестве опции. Затем второй аргумент определяется как путь поиска. Если список опций содержит `R`, используйте рекурсивный каталог. В противном случае не используйте рекурсивный поиск каталога.

17.3. Обратитесь к документации по команде `dir` или `ls` и используйте как можно больше опций в новой улучшенной версии листинга 17.8.

Что еще почитать?

- «Windows NT File System Internals: A Developer's Guide», Rajeev Nagar (O'Reilly, 1997)
- «The Boost C++ Libraries», 2nd Edition, Boris Schäling (XML Press, 2014)
- «Linux API. Исчерпывающее руководство», Майкл Керриск (Питер, 2018)

18 Алгоритмы

В этом и состоит суть программирования. К тому времени как вы разобьете сложную мысль на мелкие компоненты, легко считываемые даже глупой машиной, вы, безусловно, и сами во всем разберетесь.

Дуглас Адамс, «Детективное агентство Дирка Джентли»

Алгоритм — это процедура для решения класса задач. Библиотеки stdlib и Boost содержат множество алгоритмов, которые вы можете использовать в своих программах. Поскольку много умных людей потратили кучу времени на то, чтобы эти алгоритмы были правильными и эффективными, не нужно пытаться, например, написать собственный алгоритм сортировки.

Поскольку эта глава охватывает почти весь набор алгоритмов stdlib, она длинная. Тем не менее отдельные представления алгоритмов являются краткими. При первом чтении стоит просмотреть каждый раздел, чтобы изучить широкий спектр доступных алгоритмов. Не пытайтесь запомнить их. Изучайте задачи, которые можно решить с их помощью, когда будете писать код в будущем. Таким образом, когда нужно будет использовать алгоритм, вы скажете: «Подождите, разве кто-то уже не изобрел это колесо?»

Прежде чем вы начнете работать с алгоритмами, нужно немного разбираться в сложности и параллелизме. Эти две алгоритмические характеристики являются основными факторами, влияющими на производительность.

Сложность алгоритмов

Сложность алгоритма описывает сложность вычислительной задачи. Один из способов измерить эту сложность — использовать *обозначения Бахманна—Ландау*, или *«О большое»* (О большое). Обозначение О большое характеризует функции в соответствии с ростом вычислений относительно размера ввода. Это обозначение включает только главный член функции сложности. *Главный член* — тот, что растет быстрее всего при увеличении размера ввода.

Например, алгоритм, сложность которого увеличивается примерно на фиксированную величину для каждого дополнительного входного элемента, имеет обозначение О большое **O(N)**, в то время как алгоритм, сложность которого не изменяется при заданном дополнительном входе, имеет обозначение О большое, равное **O(1)**.

В этой главе описываются алгоритмы stdlib, которые подразделяются на пять классов сложности, как показано в следующем списке. Чтобы дать некоторое представление о том, как масштабируются эти алгоритмы, каждый класс указан в нотации О большое и с предположением о том, сколько дополнительных операций потребовалось бы из-за главного члена, когда ввод увеличивается с 1000 до 10 000 элементов. Каждый пример предоставляет операцию с данным классом сложности, где N — количество элементов, участвующих в операции:

- **Постоянное время O(1).** Никаких дополнительных вычислений. Примером является определение размера `std::vector`;
- **Логарифмическое время O(log N).** Около одного дополнительного вычисления. Примером является поиск элемента в `std::set`;
- **Линейное время O(N).** Около 9000 дополнительных вычислений. Пример — суммирование всех элементов в коллекции;
- **Квазилинейное время O(N log N).** Около 37 000 дополнительных вычислений. Примером является быстрая сортировка, широко используемый алгоритм сортировки;
- **Квадратичное время O(N2).** Около 99 000 000 дополнительных вычислений. Пример — сравнение всех элементов в коллекции со всеми элементами в другой коллекции.

Целая область computer science посвящена классификации вычислительных задач в соответствии с их сложностью, поэтому тема является актуальной. В этой главе упоминается сложность каждого алгоритма в зависимости от того, как размер целевой последовательности влияет на объем требуемой работы. На практике следует проверять производительность, чтобы определить, имеет ли алгоритм подходящие свойства масштабирования. Но эти классы сложности могут дать представление о том, насколько затратен конкретный алгоритм.

Политика выполнения

Некоторые алгоритмы, которые обычно называют *параллельными алгоритмами*, могут разделять алгоритм так, чтобы независимые объекты могли одновременно работать над различными частями проблемы. Многие алгоритмы stdlib позволяют определять параллелизм с *политикой выполнения*. Политика выполнения указывает допустимый параллелизм для алгоритма. С точки зрения stdlib, алгоритм может выполняться либо *последовательно*, либо *параллельно*. Последовательный алгоритм может иметь только один объект, работающий над проблемой в данный момент времени; параллельный алгоритм может иметь множество объектов, работающих совместно для решения проблемы.

Кроме того, параллельные алгоритмы могут быть *векторизованными* или *невекторизованными*. Векторизованные алгоритмы позволяют объектам выполнять работу в неопределенном порядке, даже позволяя одному объекту работать над несколькими частями проблемы одновременно. Например, алгоритм, который требует синхронизации между объектами, обычно не является векторизованным, поскольку один и тот же объект может пытаться получить блокировку несколько раз, что приводит к взаимоблокировке.

В заголовке `<execution>` существует три политики выполнения:

- `std::execute::seq` определяет последовательное (не параллельное) выполнение;
- `std::execute::par` определяет параллельное выполнение;
- `std::execute::par_unseq` определяет параллельное *и* векторизованное выполнение.

Для тех алгоритмов, которые поддерживают политику выполнения, по умолчанию используется `seq`, это означает, что стоит выбрать параллелизм и связанные с ним преимущества в производительности. Обратите внимание, что стандарт C++ не определяет точное значение этих политик выполнения, потому что разные платформы по-разному обрабатывают параллелизм. При предоставлении непоследовательной политики выполнения вы просто заявляете, что этот алгоритм безопасен для распараллеливания.

В главе 19 вы познакомитесь с политиками выполнения более подробно. Пока просто отметьте, что некоторые алгоритмы допускают параллелизм.

ПРЕДУПРЕЖДЕНИЕ

Описания алгоритмов в этой главе неполные. Они содержат достаточно информации, чтобы дать хорошее представление о многих алгоритмах, доступных в стандартной библиотеке. Полагаю, что после определения алгоритма, который соответствует вашим потребностям, вы посмотрите один из ресурсов в разделе «Что еще почитать?» в конце этой главы. Алгоритмы, которые принимают необязательную политику выполнения, часто предъявляют различные требования, когда предоставляются политики не по умолчанию, особенно когда речь идет об итераторах. Например, если алгоритм обычно принимает итератор ввода, использование политики выполнения, как правило, заставляет алгоритм требовать вместо этого прямых итераторов. Перечисление этих различий удлинило бы и без того громадную главу, поэтому описания здесь опущены.

КАК ИСПОЛЬЗОВАТЬ ЭТУ ГЛАВУ

Эта глава представляет собой краткий справочник, содержащий более 50 алгоритмов. Описание алгоритмов достаточно сжато. Каждый алгоритм начинается с краткого описания. Описание декларации функции алгоритма сопровождается объяснением каждого аргумента. Объявление отображает необязательные аргументы в скобках. Листинг отображает алгоритмическую сложность и заканчивается неисчерпывающим, но показательным примером использования алгоритма. Почти все примеры в этой главе являются юнит-тестами и неявно включают в себя следующее:

```
#include "catch.hpp"
#include <vector>
#include <string>

using namespace std;
```

Обратитесь к соответствующему заголовку `<algorithm>` за подробностями об алгоритмах, если в этом возникнет необходимость.

Операции, не изменяющие последовательность

Операция, не изменяющая последовательность, — это алгоритм, который выполняет вычисления над последовательностью, но не изменяет последовательность каким-либо образом. Можно представить это как алгоритмы `const`. Каждый алгоритм, описанный в этом разделе, находится в заголовке `<algorithm>`.

all_of

Алгоритм `all_of` определяет, соответствует ли каждый элемент последовательности некоторым пользовательским критериям.

Алгоритм возвращает `true`, если целевая последовательность пуста или если `pred` имеет значение `true` для *всех* элементов в последовательности; в противном случае возвращается `false`.

```
bool all_of([ep], ipt_begin, ipt_end, pred);
```

Аргументы

- Необязательная политика выполнения `std::execution`, `ep` (по умолчанию: `std::execution::seq`).
- Пара объектов `InputIterator`, `ipt_begin` и `ipt_end`, представляющих целевую последовательность.
- Унарный предикат `pred`, который принимает элемент из целевой последовательности.

Сложность

Линейная. Алгоритм вызывает `pred` не более чем `distance(ipt_begin, ipt_end)` раз.

Примеры

```
#include <algorithm>

TEST_CASE("all_of") {
  vector<string> words{ "Auntie", "Anne's", "alligator" }; ❶
  const auto starts_with_a =
    [](const auto& word❷) {
      if (word.empty()) return false; ❸
      return word[0] == 'A' || word[0] == 'a'; ❹
    };
  REQUIRE(all_of(words.cbegin(), words.cend(), starts_with_a)); ❺
  const auto has_length_six = [](const auto& word) {
    return word.length() == 6; ❻
  };
  REQUIRE_FALSE(all_of(words.cbegin(), words.cend(), has_length_six)); ❼
}
```

После создания `vector`, содержащего объекты `string` под названием `words` ❶, создается лямбда-предикат `start_with_a`, который принимает один объект с названием `word` ❷. Если `word` пустой, `start_with_a` возвращает `false` ❸; в противном случае он возвращает `true`, если `word` начинается с `a` или `A` ❹. Поскольку все элементы `word` начинаются либо с a, либо с A, `all_of` возвращает `true`, когда применяет `has_length_six` к `words` ❺.

Во втором примере создается предикат `has_length_six`, который возвращает `true`, только если `word` имеет длину шесть ❻. Поскольку длина `alligator` не равна шести, `all_of` возвращает `false` при применении `has_length_six` к `words` ❼.

any_of

Алгоритм `any_of` определяет, соответствует ли какой-либо элемент в последовательности некоторым заданным пользователем критериям.

Алгоритм возвращает `false`, если целевая последовательность пуста или если `pred` имеет значение `true` для любого элемента в последовательности; в противном случае возвращается `false`.

```
bool any_of([ep], ipt_begin, ipt_end, pred);
```

Аргументы

- Необязательная политика выполнения `std::execution`, `ep` (по умолчанию: `std::execution::seq`).
- Пара объектов `InputIterator`, `ipt_begin` и `ipt_end`, представляющих целевую последовательность.
- Унарный предикат `pred`, который принимает элемент из целевой последовательности.

Сложность

Линейная. Алгоритм вызывает `pred` не более чем `distance(ipt_begin, ipt_end)` раз.

Примеры

```
#include <algorithm>

TEST_CASE("any_of") {
  vector<string> words{ "Barber", "baby", "bubbles" }; ❶
  const auto contains_bar = [](const auto& word) {
    return word.find("Bar") != string::npos;
  }; ❷
  REQUIRE(any_of(words.cbegin(), words.cend(), contains_bar)); ❸

  const auto is_empty = [](const auto& word) { return word.empty(); }; ❹
  REQUIRE_FALSE(any_of(words.cbegin(), words.cend(), is_empty)); ❺
}
```

После создания `vector`, содержащего объекты `string` под названием `words` ❶, создается лямбда-предикат `contains_bar`, который принимает один объект с названием `word` ❷. Если `word` содержит подстроку `Bar`, возвращается `true`; в противном случае возвращается `false`. Поскольку `Barber` содержит `Bar`, `any_of` возвращает `true` при применении `contains_bar` ❸.

Во втором примере создается предикат `is_empty`, который возвращает `true`, только если `word` пустой ❹. Поскольку ни один `word` не пустой, `any_of` возвращает `false` при применении `is_empty` к `words` ❺.

none_of

Алгоритм `none_of` определяет, что ни один элемент в последовательности не соответствует некоторым заданным пользователем критериям.

Алгоритм возвращает значение `true`, если целевая последовательность пуста или если `pred` имеет значение `true` при *отсутствии* элементов в последовательности; в противном случае возвращается `false`.

```
bool none_of([ep], ipt_begin, ipt_end, pred);
```

Аргументы

- Необязательная политика выполнения `std::execution`, `ep` (по умолчанию: `std::execution::seq`).
- Пара объектов `InputIterator`, `ipt_begin` и `ipt_end`, представляющих целевую последовательность.
- Унарный предикат `pred`, который принимает элемент из целевой последовательности.

Сложность

Линейная. Алгоритм вызывает `pred` не более чем `distance(ipt_begin, ipt_end)` раз.

Примеры

```
#include <algorithm>

TEST_CASE("none_of") {
  vector<string> words{ "Camel", "on", "the", "ceiling" }; ❶
  const auto is_hump_day = [](const auto& word) {
    return word == "hump day";
  }; ❷
  REQUIRE(none_of(words.cbegin(), words.cend(), is_hump_day)); ❸

  const auto is_definite_article = [](const auto& word) {
    return word == "the" || word == "ye";
  }; ❹
  REQUIRE_FALSE(none_of(words.cbegin(), words.cend(), is_definite_article)); ❺
}
```

После создания вектора, содержащего объекты строки `words` ❶, создается лямбда-предикат `is_hump_day`, который принимает один объект с именем `word` ❷. Если `word` равен `hump day`, возвращается `true`; в противном случае возвращается `false`. Поскольку `words` не содержит `hump day`, `none_of` возвращает `true` при применении`is_hump_day` ❸.

Во втором примере создается предикат `is_definite_article`, который возвращает `true`, только если `word` является определенной статьей ❹. Поскольку это определенная статья, `none_of` возвращает `false` при применении `is_definite_article` к `words` ❺.

for_each

Алгоритм `for_each` применяет некоторую пользовательскую функцию к каждому элементу в последовательности.

Алгоритм применяет `fn` к каждому элементу целевой последовательности. Хотя `for_each` считается операцией, не изменяющей последовательность, если `ipt_begin` является изменяемым итератором, `fn` может принимать не-`const` аргумент. Любые значения, которые возвращает `fn`, игнорируются.

Если опустить `ep`, `for_each` вернет `fn`. В противном случае `for_each` возвращает `void`.

```
for_each([ep], ipt_begin, ipt_end, fn);
```

Аргументы

- Необязательная политика выполнения `std::execution`, `ep` (по умолчанию: `std::execution::seq`).

- Пара объектов `InputIterator`, `ipt_begin` и `ipt_end`, представляющих целевую последовательность.
- Унарная функция `fn`, которая принимает элемент из целевой последовательности.

Сложность

Линейная. Алгоритм вызывает `fn` точно `distance(ipt_begin, ipt_end)` раз.

Дополнительные требования

- `fn` должна быть перемещаемой, если пропустить `ep`.
- `fn` должна быть копируемой, если предоставить `ep`.

Пример

```
#include <algorithm>

TEST_CASE("for_each") {
  vector<string> words{ "David", "Donald", "Doo" }; ❶
  size_t number_of_Ds{}; ❷
  const auto count_Ds = [&number_of_Ds❸](const auto& word❹) {
    if (word.empty()) return; ❺
    if (word[0] == 'D') ++number_of_Ds; ❻
  };
  for_each(words.cbegin(), words.cend(), count_Ds); ❼
  REQUIRE(3 == number_of_Ds); ❽
}
```

После создания вектора, содержащего объекты строки `words` ❶, и переменной-счетчика `number_of_Ds` ❷ создается лямбда-предикат `count_Ds`, который захватывает ссылку на `number_of_Ds` ❸ и принимает один объект с названием `word` ❹. Если `word` пустой, выполнение завершится, возвращается true ❺; в противном случае, если первая буква `word` — `D`, увеличивается `number_of_Ds` ❻.

Затем `for_each` используется для итерации по каждому `word`, передавая каждый из них в `count_Ds` ❼. Результатом является то, что `number_of_Ds` равен трем ❽.

for_each_n

Алгоритм `for_each_n` применяет некоторую пользовательскую функцию к каждому элементу в последовательности.

Алгоритм применяет `fn` к каждому элементу целевой последовательности. Хотя `for_each_n` считается операцией, не изменяющей последовательность, если `ipt_begin` является изменяемым итератором, `fn` может принимать не-`const` аргумент. Любые значения, которые возвращает `fn`, игнорируются. Возвращается `ipt_begin+n`.

```
InputIterator for_each_n([ep], ipt_begin, n, fn);
```

Аргументы

- Необязательная политика выполнения `std::execution`, `ep` (по умолчанию: `std::execution::seq`).
- `InputIterator ipt_begin`, представляющий первый элемент целевой последовательности.
- Целое число `n`, представляющее желаемое количество итераций, — полуоткрытый диапазон, представляющий целевую последовательность от `ipt_begin` до `ipt_begin+n` (`Size` — это шаблонный тип `n`).
- Унарная функция `fn`, которая принимает элемент из целевой последовательности.

Сложность

Линейная. Алгоритм вызывает `fn` ровно `n` раз.

Дополнительные требования

- `fn` должна быть перемещаемой, если пропустить `ep`.
- `fn` должна быть копируемой, если предоставить `ep`.
- `n` должен быть неотрицательным.

Пример

```
#include <algorithm>

TEST_CASE("for_each_n") {
  vector<string> words{ "ear", "egg", "elephant" }; ❶
  size_t characters{}; ❷
  const auto count_characters = [&characters❸](const auto& word❹) {
    characters += word.size(); ❺
  };
  for_each_n(words.cbegin(), words.size(), count_characters); ❻
  REQUIRE(14 == characters); ❼
}}
```

После создания вектора, содержащего объекты строки `words` ❶, и переменной-счетчика `characters` ❷ создается лямбда-предикат `count_characters`, который захватывает ссылку на `characters` ❸ и принимает один объект с названием `word` ❹. Лямбда-предикат добавляет длину `word` к `characters` ❺.

Затем `for_each` используется для итерации по каждому `word`, передавая каждый из них в `count_characters` ❻. Результатом является то, что `characters` равен 14 ❼.

find, find_if и find_if_not

Алгоритмы `find`, `find_if` и `find_if_not` находят первый элемент в последовательности, соответствующий некоторым заданным пользователем критериям.

Эти алгоритмы возвращают `InputIterator`, указывающий на значение соответствия первого элемента целевой последовательности (`find`), что приводит к истинному результату при вызове с помощью `pred(find_if)` или к ложному результату при вызове с помощью `pred(find_if_not)`.

Если алгоритм не находит соответствия, возвращается `ipt_end`.

```
InputIterator find([ep], ipt_begin, ipt_end, value);
InputIterator find_if([ep], ipt_begin, ipt_end, pred);
InputIterator find_if_not([ep], ipt_begin, ipt_end, pred);
```

Аргументы

- Необязательная политика выполнения `std::execution`, `ep` (по умолчанию: `std::execution::seq`).
- Пара объектов `InputIterator`, `ipt_begin` и `ipt_end`, представляющих целевую последовательность.
- `const` ссылка `value`, которая тождественно сравнима с базовым типом целевой последовательности (`find`), или предикат, который принимает один аргумент с базовым типом целевой последовательности (`find_if` и `find_if_not`).

Сложность

Линейная. Алгоритм выполняет максимум `distance(ipt_begin, ipt_end)` сравнений (`find`) или вызовов `pred` (`find_if` и `find_if_not`).

Примеры

```
#include <algorithm>

TEST_CASE("find find_if find_if_not") {
  vector<string> words{ "fiffer", "feffer", "feff" }; ❶
  const auto find_result = find(words.cbegin(), words.cend(), "feff"); ❷
  REQUIRE(*find_result == words.back()); ❸

  const auto defends_digital_privacy = [](const auto& word) {
    return string::npos != word.find("eff"); ❹
  };
  const auto find_if_result = find_if(words.cbegin(), words.cend(),
                                      defends_digital_privacy); ❺
  REQUIRE(*find_if_result == "feffer"); ❻

  const auto find_if_not_result = find_if_not(words.cbegin(), words.cend(),
                                              defends_digital_privacy); ❼
  REQUIRE(*find_if_not_result == words.front()); ❽
}
```

После создания вектора, содержащего объекты строки `words` ❶ используется `find` для определения местоположения `feff` ❷, который находится в конце `words` ❸. Затем создается предикат `defnds_digital_privacy`, который возвращает `true`, если слово содержит буквы `eff` ❹. Затем используется `find_if`, чтобы найти первую строку

в words, которая содержит eff ❺, feffer ❻. Наконец, используется find_if_not, чтобы применить defnds_digital_privacy к words ❼, который возвращает первый элемент fiffer (потому что он не содержит eff) ❽.

find_end

Алгоритм find_end находит последнее вхождение подпоследовательности.

Если алгоритм не находит такой последовательности, он возвращает fwd_end1. Если find_end находит подпоследовательность, он возвращает ForwardIterator, указывающий на первый элемент последней соответствующей подпоследовательности.

```
InputIterator find_end([ep], fwd_begin1, fwd_end1,
                       fwd_begin2, fwd_end2, [pred]);
```

Аргументы

- Необязательная политика выполнения std::execution, ep (по умолчанию: std::execution::seq).
- Две пары ForwardIterators, fwd_begin1/fwd_end1 и fwd_begin2/fwd _end2, представляющие целевые последовательности 1 и 2.
- Необязательный двоичный предикат pred для сравнения, равны ли два элемента.

Сложность

Квадратичная. Алгоритм выполняет максимум следующее количество сравнений или вызовов pred:

```
distance(fwd_begin2, fwd_end2) * (distance(fwd_begin1, fwd_end1) -
                                  distance(fwd_begin2, fwd_end2) + 1)
```

Примеры

```
#include <algorithm>

TEST_CASE("find_end") {
  vector<string> words1{ "Goat", "girl", "googoo", "goggles" }; ❶
  vector<string> words2{ "girl", "googoo" }; ❷
  const auto find_end_result1 = find_end(words1.cbegin(), words1.cend(),
                                         words2.cbegin(), words2.cend()); ❸
  REQUIRE(*find_end_result1 == words1[1]); ❹

  const auto has_length = [](const auto& word, const auto& len) {
    return word.length() == len; ❺
  };
  vector<size_t> sizes{ 4, 6 }; ❻
  const auto find_end_result2 = find_end(words1.cbegin(), words1.cend(),
                                         sizes.cbegin(), sizes.cend(),
                                         has_length); ❼
  REQUIRE(*find_end_result2 == words1[1]); ❽
}
```

После создания вектора, содержащего объекты строк `words1` ❶ и `words2` ❷, вызывается `find_end`, чтобы определить, какой элемент в `words1` начинает подпоследовательность, равную `words2` ❸. Результатом является `find_end_result1`, который равен элементу `girl` ❹.

Затем создается лямбда-предикат `has_length`, который принимает два аргумента, `word` и `len`, и возвращает `true`, если `word.length()` равен `len` ❺. Создается `vector` объектов `size_t` под названием `sizes` ❻ и вызывается `find_end` с `words1`, `sizes` и `has_length` ❼. Результат `find_end_result2` указывает на первый элемент в `words1`, имеющий длину 4, с последующим словом с длиной 6. Поскольку `girl` имеет длину 4, а `googoo` имеет длину 6, `find_end_result2` указывает на `girl` ❽.

find_first_of

Алгоритм `find_first_of` находит первое вхождение в последовательности 1, равное некоторому элементу в последовательности 2.

Если задать `pred`, алгоритм находит первое вхождение `i` в последовательности `1`, где для некоторого `j` в последовательности 2 `pred(i, j)` имеет значение `true`.

Если `find_first_of` не находит такой последовательности, он возвращает `ipt_end1`. Если `find_first_of` находит подпоследовательность, он возвращает `InputIterator`, указывающий на первый элемент первой соответствующей подпоследовательности. (Обратите внимание, что, если `ipt_begin1` также является `ForwardIterator`, `find_first_of` вместо этого возвращает `ForwardIterator`.)

```
InputIterator find_first_of([ep], ipt_begin1, ipt_end1,
                            fwd_begin2, fwd_end2, [pred]);
```

Аргументы

- Необязательная политика выполнения `std::execution`, `ep` (по умолчанию: `std::execution::seq`).
- Пара объектов `InputIterator`, `ipt_begin1/ipt_end1`, представляющая целевую последовательность 1.
- Пара `ForwardIterators`, `fwd_begin2/fwd_end2`, представляющая целевую последовательность 2.
- Необязательный двоичный предикат `pred` для сравнения, равны ли два элемента.

Сложность

Квадратичная. Алгоритм выполняет максимум следующее количество сравнений или вызовов `pred`:

```
distance(ipt_begin1, ipt_end1) * distance(fwd_begin2, fwd_end2)
```

Пример

```
#include <algorithm>

TEST_CASE("find_first_of") {
  vector<string> words{ "Hen", "in", "a", "hat" }; ❶
  vector<string> indefinite_articles{ "a", "an" }; ❷
  const auto find_first_of_result = find_first_of(words.cbegin(),
                                                  words.cend(),
                                                  indefinite_articles.cbegin(),
                                                  indefinite_articles.cend()); ❸
  REQUIRE(*find_first_of_result == words[2]); ❹
}
```

После создания вектора, содержащего объекты строки `words` ❶ и лямбда-предиката `indefinite_articles` ❷, вызывается `find_first_of`, чтобы определить, какой элемент в words1 начинает подпоследовательность, равную `indefinite_articles` ❸. Результатом является `find_first_of_result`, который равен элементу `a` ❹.

adjacent_find

Алгоритм `adjacent_find` находит первый повтор в последовательности.

Алгоритм находит первое вхождение в целевой последовательности, где два смежных элемента равны или где, если задать `pred`, алгоритм находит первый элемент вхождения `i` в последовательности, где `pred(i, i+1)` имеет значение `true`.

Если `adjacent_find` не находит такого элемента, он возвращает `fwd_end`. Если `adjacent_find` обнаруживает такой элемент, он возвращает `ForwardIterator`, указывающий на него.

```
ForwardIterator adjacent_find([ep], fwd_begin, fwd_end, [pred]);
```

Аргументы

- Необязательная политика выполнения `std::execution`, `ep` (по умолчанию: `std::execution::seq`).
- Пара `ForwardIterators`, `fwd_begin/fwd_end`, представляющая целевую последовательность.
- Необязательный двоичный предикат `pred` для сравнения, равны ли два элемента.

Сложность

Линейная. Когда политика выполнения не указана, алгоритм выполняет не более следующего числа сравнений или вызовов `pred`:

```
min(distance(fwd_begin, i)+1, distance(fwd_begin, fwd_end)-1),
```

где **`i`** — индекс возвращаемого значения.

Пример

```
#include <algorithm>
TEST_CASE("adjacent_find") {
  vector<string> words{ "Icabod", "is", "itchy" }; ❶
  const auto first_letters_match = [](const auto& word1, const auto& word2) { ❷
    if (word1.empty() || word2.empty()) return false;
    return word1.front() == word2.front();
  };
  const auto adjacent_find_result = adjacent_find(words.cbegin(), words.cend(),
                                                  first_letters_match); ❸
  REQUIRE(*adjacent_find_result == words[1]); ❹
}
```

После создания вектора, содержащего объекты строки `words` ❶, создается лямбда-предикат `first_letters_match`, который принимает два слова и проверяет, не начинаются ли они с одной и той же первой буквы ❷. Вызывается `adjacent_find`, чтобы определить, какой элемент начинается с той же буквы, что и первая буква в целевой последовательности ❸. Результат, `adjacent_find_result` ❹, равен `is`, поскольку он разделяет первую букву с `itchy` ❹.

count

Алгоритм `count` считает элементы в последовательности, соответствующие определенным пользователем критериям.

Алгоритм возвращает количество элементов `i` в целевой последовательности, где `pred(i)` равно `true` или где `value == i`. Обычно `DifferenceType` означает `size_t`, но это зависит от реализации `InputIterator`. `count` используется, когда нужно подсчитать вхождения определенного значения, а `count_if` — когда есть более сложный предикат, который нужно использовать для сравнения.

```
DifferenceType count([ep], ipt_begin, ipt_end, value);
DifferenceType count_if([ep], ipt_begin, ipt_end, pred);
```

Аргументы

- Необязательная политика выполнения `std::execution`, `ep` (по умолчанию: `std::execution::seq`).
- Пара объектов `InputIterator`, `ipt_begin/ipt_end`, представляющих целевую последовательность.
- Либо `value`, либо унарный предикат `pred`, чтобы оценить, должен ли элемент x в целевой последовательности быть посчитан как подходящий.

Сложность

Линейная. Когда политика выполнения не указана, алгоритм выполняет максимум сравнений `distance(ipt_begin, ipt_end)` или вызовов `pred`.

Примеры

```
#include <algorithm>
TEST_CASE("count") {
  vector<string> words{ "jelly", "jar", "and", "jam" }; ❶
  const auto n_ands = count(words.cbegin(), words.cend(), "and"); ❷
  REQUIRE(n_ands == 1); ❸

  const auto contains_a = [](const auto& word) { ❹
    return word.find('a') != string::npos;
  };
  const auto count_if_result = count_if(words.cbegin(), words.cend(),
                                        contains_a); ❺
  REQUIRE(count_if_result == 3); ❻
}
```

После создания вектора, содержащего объекты строки `words` ❶, вызывается `count` со значением `and` ❷. Это возвращает 1, потому что только один элемент равен `and` ❸. Затем создается лямбда-предикат с именем `contains_a`, который принимает слово и определяет, содержит ли оно `a` ❹. Вызывается `count_if`, чтобы определить, сколько слов содержат `a` ❺. Результат равен 3, потому что три элемента содержат `a` ❻.

mismatch

Алгоритм `mismatch` находит первое несоответствие в двух последовательностях.

Алгоритм находит первую пару несовпадающих элементов **`i`**, **`j`** из последовательности 1 и последовательности 2. В частности, он находит первый индекс **`n`** такой, что **`i = (ipt_begin1 + n); j = (ipt_begin2 + n);`** и **`i != j`** или **`pred(i, j) == false`**.

Типы итераторов в возвращаемой `pair` равны типам `ipt_begin1` и `ipt_begin2`.

```
pair<Itr, Itr> mismatch([ep], ipt_begin1, ipt_end1,
                        ipt_begin2, [ipt_end2], [pred]);
```

Аргументы

- Необязательная политика выполнения `std::execution`, `ep` (по умолчанию: `std::execution::seq`).
- Две пары `InputIterators`, `ipt_begin1/ipt_end1` и `ipt_begin2/ipt _end2`, представляющие целевые последовательности 1 и 2. Если не предоставить `ipt_end2`, длина последовательности 1 будет подразумевать длину последовательности 2.
- Необязательный двоичный предикат `pred` для сравнения, равны ли два элемента.

Сложность

Линейная. Когда политика выполнения не указана, в худшем случае алгоритм выполняет следующее число сравнений или вызовов `pred`:

```
min(distance(ipt_begin1, ipt_end1), distance(ipt_begin2, ipt_end2))
```

Примеры

```
#include <algorithm>

TEST_CASE("mismatch") {
  vector<string> words1{ "Kitten", "Kangaroo", "Kick" }; ❶
  vector<string> words2{ "Kitten", "bandicoot", "roundhouse" }; ❷
  const auto mismatch_result1 = mismatch(words1.cbegin(), words1.cend(),
                                         words2.cbegin()); ❸
  REQUIRE(*mismatch_result1.first == "Kangaroo"); ❹
  REQUIRE(*mismatch_result1.second == "bandicoot"); ❺

  const auto second_letter_matches = [](const auto& word1,
                                        const auto& word2) { ❻
    if (word1.size() < 2) return false;
    if (word2.size() < 2) return false;
    return word1[1] == word2[1];
  };
  const auto mismatch_result2 = mismatch(words1.cbegin(), words1.cend(),
                                         words2.cbegin(), second_letter_matches); ❼
  REQUIRE(*mismatch_result2.first == "Kick"); ❽
  REQUIRE(*mismatch_result2.second == "roundhouse"); ❾
}
```

После создания двух векторов, содержащих объекты строк `words1` ❶ и `words2` ❷, они используются в качестве целевых последовательностей для `mismatch` ❸. Это возвращает пару, указывающую на элементы `Kangaroo` и `bandicoot` ❹ ❺. Затем создается лямбда-предикат с именем `second_letter_matches`, который принимает два слова и оценивает, соответствуют ли их вторые буквы ❻. `mismatch` вызывается, чтобы определить первую пару элементов с несовпадающими вторыми буквами ❼. В результате получается пара `Kick` ❽ и `roundhouse` ❾.

equal

Алгоритм `equal` определяет, равны ли две последовательности.

Алгоритм определяет, равны ли элементы последовательности 1 последовательности 2.

```
bool equal([ep], ipt_begin1, ipt_end1, ipt_begin2, [ipt_end2], [pred]);
```

Аргументы

- Необязательная политика выполнения `std::execution`, `ep` (по умолчанию: `std::execution::seq`).
- Две пары `InputIterators`, `ipt_begin1/ipt_end1` и `ipt_begin2/ipt_end2`, представляющие целевые последовательности 1 и 2. Если не предоставить `ipt_end2`, длина последовательности 1 подразумевает длину последовательности 2.
- Необязательный двоичный предикат `pred` для сравнения, равны ли два элемента.

Сложность

Линейная. Когда политика выполнения не указана, в худшем случае алгоритм выполняет следующее число сравнений или вызовов `pred`:

```
min(distance(ipt_begin1, ipt_end1), distance(ipt_begin2, ipt_end2))
```

Примеры

```
#include <algorithm>

TEST_CASE("equal") {
  vector<string> words1{ "Lazy", "lion", "licks" }; ❶
  vector<string> words2{ "Lazy", "lion", "kicks" }; ❷
  const auto equal_result1 = equal(words1.cbegin(), words1.cend(),
                                   words2.cbegin()); ❸
  REQUIRE_FALSE(equal_result1); ❹

  words2[2] = words1[2]; ❺
  const auto equal_result2 = equal(words1.cbegin(), words1.cend(),
                                   words2.cbegin()); ❻
  REQUIRE(equal_result2); ❼
}
```

После создания двух векторов, содержащих объекты строк `words1` и `words2` ❶ ❷, они используются в качестве целевых последовательностей для `equal` ❸. Поскольку их последние элементы, `lick` и `kick`, не равны, `equal_result1` равен `false` ❹. После задания третьего элемента `words2` равным третьему элементу `words1` ❺ снова вызывается `equal` с теми же аргументами ❻. Поскольку последовательности теперь идентичны, `equal_result2` имеет значение `true` ❼.

is_permutation

Алгоритм `is_permutation` определяет, являются ли две последовательности перестановками, то есть они содержат одинаковые элементы, но потенциально в другом порядке.

Алгоритм определяет, существует ли некоторая перестановка последовательности 2, так что элементы последовательности 1 равны перестановкам.

```
bool is_permutation([ep], fwd_begin1, fwd_end1, fwd_begin2, [fwd_end2], [pred]);
```

Аргументы

- Необязательная политика выполнения `std::execution`, `ep` (по умолчанию: `std::execution::seq`).
- Две пары `ForwardIterators`, `fwd_begin1`/`fwd_end1` и `fwd_begin2`/`fwd_end2`, представляющие целевые последовательности 1 и 2. Если не предоставить `fwd_end2`, длина последовательности 1 подразумевает длину последовательности 2.
- Необязательный двоичный предикат `pred` для сравнения, равны ли два элемента.

Сложность

Квадратичная. Когда политика выполнения не задана, в худшем случае алгоритм выполняет следующее число сравнений или вызовов `pred`:

```
distance(fwd_begin1, fwd_end1) * distance(fwd_begin2, fwd_end2)
```

Пример

```
#include <algorithm>

TEST_CASE("is_permutation") {
  vector<string> words1{ "moonlight", "mighty", "nice" }; ❶
  vector<string> words2{ "nice", "moonlight", "mighty" }; ❷
  const auto result = is_permutation(words1.cbegin(), words1.cend(),
                                     words2.cbegin()); ❸
  REQUIRE(result); ❹
}
```

После создания двух векторов, содержащих объекты строк `words1` и `words2` ❶ ❷, они используются в качестве целевых последовательностей для `is_permutation` ❸. Поскольку `words2` является перестановкой `words1`, `is_permutation` возвращает `true` ❹.

ПРИМЕЧАНИЕ

Заголовок <algorithm> также содержит next_permutation и prev_permutation для управления диапазоном элементов, чтобы можно было генерировать перестановки. См. [alg.permutation.generators].

search

Алгоритм `search` находит подпоследовательность.

Алгоритм находит последовательность 2 в последовательности 1. Другими словами, он возвращает первый итератор `i` в последовательности 1, так что для каждого неотрицательного целого числа `n` `*(i + n)` было равно `*(fwd_begin2 + n)` или, если предоставить предикат `pred(*(i + n), *(fwd_begin2 + n))`, являлось `true`. Алгоритм поиска возвращает `fwd_begin1`, если последовательность 2 пуста, или `fwd_begin2`, если подпоследовательность не найдена. Это отличается от `find`, потому что ищется подпоследовательность, а не отдельный элемент.

```
ForwardIterator search([ep], fwd_begin1, fwd_end1,
                            fwd_begin2, fwd_end2, [pred]);
```

Аргументы

- Необязательная политика выполнения `std::execution`, `ep` (по умолчанию: `std::execution::seq`).

- Две пары `ForwardIterators`, `fwd_begin1/fwd_end1` и `fwd_begin2/fwd_end2`, представляющие целевые последовательности 1 и 2.
- Необязательный двоичный предикат `pred` для сравнения, равны ли два элемента.

Сложность

Квадратичная. Когда политика выполнения не задана, в худшем случае алгоритм выполняет следующее число сравнений или вызовов `pred`:

```
distance(fwd_begin1, fwd_end1) * distance(fwd_begin2, fwd_end2)
```

Примеры

```
#include <algorithm>

TEST_CASE("search") {
  vector<string> words1{ "Nine", "new", "neckties", "and",
                         "a", "nightshirt" }; ❶
  vector<string> words2{ "and", "a", "nightshirt" }; ❷
  const auto search_result_1 = search(words1.cbegin(), words1.cend(),
                                      words2.cbegin(), words2.cend()); ❸
  REQUIRE(*search_result_1 == "and"); ❹

  vector<string> words3{ "and", "a", "nightpant" }; ❺
  const auto search_result_2 = search(words1.cbegin(), words1.cend(),
                                      words3.cbegin(), words3.cend()); ❻
  REQUIRE(search_result_2 == words1.cend()); ❼
}
```

После создания двух векторов, содержащих объекты строк `words1` ❶ и `words2` ❷, они используются в качестве целевых последовательностей для `search` ❸. Поскольку `words2` является подпоследовательностью `words1`, `search` возвращает итератор, указывающий на `and` ❹. `vector`, содержащий строковые объекты `words3` ❺, вместо `nightshirt` содержит слово `nightpant`, поэтому при вызове `search` с ним вместо `words2` ❻ получается конечный итератор `words1` ❼.

search_n

Алгоритм `search_n` находит подпоследовательность, содержащую идентичные последовательные значения.

Алгоритм ищет количество последовательных значений в последовательности и возвращает итератор, указывающий на первое значение, или возвращает `fwd_end`, если такая подпоследовательность не найдена. Это отличается от `adjacent_find`, потому что ищется подпоследовательность, а не один элемент.

```
ForwardIterator search_n([ep], fwd_begin, fwd_end, count, value, [pred]);
```

Аргументы

- Необязательная политика выполнения `std::execution`, `ep` (по умолчанию: `std::execution::seq`).
- Пара `ForwardIterators`, `fwd_begin/fwd_end`, представляющая целевую последовательность.
- Значение целого числа, представляющее количество последовательных совпадений, которое нужно найти.
- Значение, представляющее элемент, который нужно найти.
- Необязательный двоичный предикат `pred` для сравнения, равны ли два элемента.

Сложность

Линейная. Когда политика выполнения не указана, в худшем случае алгоритм выполняет `distance(fwd_begin, fwd_end)` сравнений или вызовов `pred`.

Пример

```
#include <algorithm>

TEST_CASE("search_n") {
  vector<string> words{ "an", "orange", "owl", "owl", "owl", "today" }; ❶
  const auto result = search_n(words.cbegin(), words.cend(), 3, "owl"); ❷
  REQUIRE(result == words.cbegin() + 2); ❸
}
```

После создания вектора, содержащего объекты строк `words` ❶, он используется в качестве целевой последовательности для `search_n` ❷. Поскольку `words` содержит три экземпляра слова `owl`, он возвращает итератор, указывающий на первый экземпляр ❸.

Операции, изменяющие последовательность

Операция, изменяющая последовательность, — это алгоритм, который выполняет вычисления над последовательностью и может каким-либо образом изменять ее. Каждый алгоритм, описанный в этом разделе, находится в заголовке `<algorithm>`.

copy

Алгоритм `copy` копирует одну последовательность в другую.

Алгоритм копирует целевую последовательность в результат и возвращает конечный итератор принимающей последовательности. Вы несете ответственность за обеспечение того, чтобы результат представлял последовательность с достаточным пространством для хранения целевой последовательности.

```
OutputIterator copy([ep], ipt_begin, ipt_end, result);
```

Аргументы

- Необязательная политика выполнения `std::execution`, `ep` (по умолчанию: `std::execution::seq`).
- Пара объектов `InputIterator`, `ipt_begin` и `ipt_end`, представляющих целевую последовательность.
- Результат `OutputIterator`, который получает скопированную последовательность.

Сложность

Линейная. Алгоритм копирует элементы из целевой последовательности точно `distance(ipt_begin, ipt_end)` раз.

Дополнительные требования

Последовательности 1 и 2 не должны пересекаться, если только операция не является *копированием слева*. Например, для вектора `v` с 10 элементами `std::copy(v.begin() + 3, v.end(), v.begin())`подходит, но `std::copy(v.begin(), v.begin() + 7, v.begin() + 3)` — нет.

ПРИМЕЧАНИЕ

Вспомните `back_inserter` из раздела «Итераторы вставки», который возвращает итератор вывода, преобразующий операции записи в операции вставки в базовом контейнере.

Пример

```
#include <algorithm>

TEST_CASE("copy") {
  vector<string> words1{ "and", "prosper" }; ❶
  vector<string> words2{ "Live", "long" }; ❷
  copy(words1.cbegin(), words1.cend(), ❸
       back_inserter(words2)❹);
  REQUIRE(words2 == vector<string>{ "Live", "long", "and", "prosper" }); ❺
}
```

После создания двух векторов ❶ ❷, содержащих объекты строки, вызывается `copy` с `words1` в качестве последовательности для `copy` ❸ и `words2` в качестве целевой последовательности ❹. Результатом является то, что `words2` содержит контент `words1`, добавленный к исходному содержимому ❺.

copy_n

Алгоритм `copy_n` копирует одну последовательность в другую.

Алгоритм копирует целевую последовательность в `result` и возвращает конечный итератор принимающей последовательности. Вы несете ответственность за то, чтобы результат представлял последовательность с достаточным пространством для

хранения целевой последовательности и чтобы n представляло правильную длину целевой последовательности.

```
OutputIterator copy_n([ep], ipt_begin, n, result);
```

Аргументы

- Необязательная политика выполнения `std::execution`, `ep` (по умолчанию: `std::execution::seq`).
- Начальный итератор `ipt_begin`, представляющий начало целевой последовательности.
- Размер целевой последовательности `n`.
- Результат `OutputIterator`, который получает скопированную последовательность.

Сложность

Линейная. Алгоритм копирует элементы из целевой последовательности ровно `distance(ipt_begin, ipt_end)` раз.

Дополнительные требования

Последовательности 1 и 2 не должны содержать одинаковые объекты, если только операция не является *копированием слева*.

Пример

```
#include <algorithm>

TEST_CASE("copy_n") {
  vector<string> words1{ "on", "the", "wind" }; ❶
  vector<string> words2{ "I'm", "a", "leaf" }; ❷
  copy_n(words1.cbegin(), words1.size(), ❸
         back_inserter(words2)); ❹
  REQUIRE(words2 == vector<string>{ "I'm", "a", "leaf",
                                    "on", "the", "wind" }); ❺
}
```

После создания двух векторов, содержащих объекты строк ❶ ❷, вызывается `copy_n` с `words1` в качестве последовательности для `copy_n` ❸ и `words2` в качестве целевой последовательности ❹. Результатом является то, что `words2` содержит контент `words1`, добавленный к исходному содержимому ❺.

copy_backward

Алгоритм `copy_backward` копирует обратную последовательность из одной последовательности в другую.

Алгоритм копирует последовательность 1 в последовательность 2 и возвращает конечный итератор принимающей последовательности. Элементы копируются

в обратном порядке, но будут отображаться в целевой последовательности в исходном порядке. Вы несете ответственность за то, чтобы последовательность 1 представляла последовательность с достаточным пространством для хранения последовательности 2.

```
OutputIterator copy_backward([ep], ipt_begin1, ipt_end1, ipt_end2);
```

Аргументы

- Необязательная политика выполнения `std::execution`, `ep` (по умолчанию: `std::execution::seq`).
- Пара объектов `InputIterator`, `ipt_begin1` и `ipt_end1`, представляющих последовательность.
- `InputIterator`, `ipt_end2`, представляющий элемент сразу за последним в последовательности 2.

Сложность

Линейная. Алгоритм копирует элементы из целевой последовательности ровно `distance(ipt_begin1, ipt_end1)` раз.

Дополнительные требования

Последовательности 1 и 2 не должны пересекаться.

Пример

```
#include <algorithm>

TEST_CASE("copy_backward") {
  vector<string> words1{ "A", "man", "a", "plan", "a", "bran", "muffin" }; ❶
  vector<string> words2{ "a", "canal", "Panama" }; ❷
  const auto result = copy_backward(words2.cbegin(), words2.cend(), ❸
                                    words1.end()); ❹
  REQUIRE(words1 == vector<string>{ "A", "man", "a", "plan",
                                    "a", "canal", "Panama" }); ❺
}
```

После создания двух векторов, содержащих объекты строк ❶ ❷, вызывается `copy_backward` с `words2` в качестве последовательности для копирования ❸ и `words1` в качестве целевой последовательности ❹. В результате содержимое `word2` заменяет последние три слова `word1` ❺.

move

Алгоритм `move` перемещает одну последовательность в другую.

Алгоритм перемещает целевую последовательность и возвращает конечный итератор принимающей последовательности. Вы несете ответственность за то, чтобы целевая

последовательность представляла последовательность, содержащую как минимум столько же элементов, сколько исходная последовательность.

```
OutputIterator move([ep], ipt_begin, ipt_end, result);
```

Аргументы

- Необязательная политика выполнения `std::execution`, `ep` (по умолчанию: `std::execution::seq`).
- Пара объектов `InputIterator`, `ipt_begin` и `ipt_end`, представляющих целевую последовательность.
- `InputIterator`, результат, представляющий начало последовательности, в которую перемещаются элементы.

Сложность

Линейная. Алгоритм перемещает элементы из целевой последовательности ровно `distance(ipt_begin, ipt_end)` раз.

Дополнительные требования

- Последовательности не должны пересекаться, если не перемещаются влево.
- Типы должны быть перемещаемыми, но не обязательно копируемыми.

Пример

```
#include <algorithm>

struct MoveDetector { ❶
  MoveDetector() : owner{ true } {} ❷
  MoveDetector(const MoveDetector&) = delete;
  MoveDetector& operator=(const MoveDetector&) = delete;
  MoveDetector(MoveDetector&& o) = delete;
  MoveDetector& operator=(MoveDetector&&) { ❸
    o.owner = false;
    owner = true;
    return *this;
  }
  bool owner;
};

TEST_CASE("move") {
  vector<MoveDetector> detectors1(2); ❹
  vector<MoveDetector> detectors2(2); ❺
  move(detectors1.begin(), detectors1.end(), detectors2.begin()); ❻
  REQUIRE_FALSE(detectors1[0].owner); ❼
  REQUIRE_FALSE(detectors1[1].owner); ❽
  REQUIRE(detectors2[0].owner); ❾
  REQUIRE(detectors2[1].owner); ❿
}
```

Сначала объявляется класс `MoveDetector` ❶, который определяет конструктор по умолчанию, устанавливающий для его единственного члена `owner` значение `true` ❷. Он удаляет конструктор копирования и перемещения и оператор присваивания копии, но определяет оператор присваивания переноса, который меняет `owner` ❸.

После создания двух объектов `MoveyDetector` ❹ ❺ вызывается `move` с `detectors1` в качестве последовательности для `move` и `detectors2` в качестве последовательности назначения ❻. Результатом является то, что элементы detectors1 находятся в перемещенном состоянии ❼ ❽, а элементы `detectors1` перемещаются в `detectors2` ❾ ❿.

move_backward

Алгоритм `move_backward` перемещает одну последовательность в обратном порядке в другую.

Алгоритм перемещает последовательность 1 в последовательность 2 и возвращает итератор, указывающий на последний перемещенный элемент. Элементы перемещаются в обратном порядке, но появляются в целевой последовательности в исходном порядке. Вы несете ответственность за то, чтобы целевая последовательность представляла последовательность, содержащую как минимум столько же элементов, сколько исходная последовательность.

```
OutputIterator move_backward([ep], ipt_begin, ipt_end, result);
```

Аргументы

- Необязательная политика выполнения `std::execution`, `ep` (по умолчанию: `std::execution::seq`).
- Пара объектов `InputIterator`, `ipt_begin` и `ipt_end`, представляющих целевую последовательность.
- Результат `InputIterator`, представляющий последовательность, в которую перемещаются элементы.

Сложность

Линейная. Алгоритм перемещает элементы из целевой последовательности ровно `distance(ipt_begin, ipt_end)` раз.

Дополнительные требования

- Последовательности не должны пересекаться.
- Типы должны быть перемещаемыми, но не обязательно копируемыми.

Сначала объявляется класс `MoveDetector` ❶ (см. раздел «Перенос» для реализации). После построения двух `vector` объектов `MoveDetector` ❷ ❸ вызывается

`move` с `detectors1` в качестве последовательности для переноса и `detectors2` в качестве последовательности назначения ❹. Результатом является то, что элементы `detectors1` находятся в состоянии переноса ❺ ❻, а элементы `detectors1` перемещаются ❼ ❽.

Пример

```
#include <algorithm>

struct MoveDetector { ❶
--пропуск--
};

TEST_CASE("move_backward") {
  vector<MoveDetector> detectors1(2); ❷
  vector<MoveDetector> detectors2(2); ❸
  move_backward(detectors1.begin(), detectors1.end(), detectors2.end()); ❹
  REQUIRE_FALSE(detectors1[0].owner); ❺
  REQUIRE_FALSE(detectors1[1].owner); ❻
  REQUIRE(detectors2[0].owner); ❼
  REQUIRE(detectors2[1].owner); ❽
}
```

swap_ranges

Алгоритм `swap_ranges` меняет элементы одной последовательности на элементы другой.

Алгоритм вызывает `swap` для каждого элемента последовательности 1 и последовательности 2 и возвращает конечный итератор принимающей последовательности. Вы несете ответственность за то, чтобы целевая последовательность представляла последовательность, содержащую как минимум столько же элементов, сколько исходная последовательность.

```
OutputIterator swap_ranges([ep], ipt_begin1, ipt_end1, ipt_begin2);
```

Аргументы

- Необязательная политика выполнения `std::execution`, `ep` (по умолчанию: `std::execution::seq`).
- Пара `ForwardIterators`, `ipt_begin1` и `ipt_end1`, представляющих последовательность 1.
- `ForwardIterator`, `ipt_begin2`, представляющий начало последовательности 2.

Сложность

Линейная. Алгоритм вызывает `swap` ровно `distance(ipt_begin1, ipt_end1)` раз.

Дополнительные требования

Элементы, содержащиеся в каждой последовательности, должны быть заменяемыми.

Пример

```
#include <algorithm>

TEST_CASE("swap_ranges") {
  vector<string> words1{ "The", "king", "is", "dead." }; ❶
  vector<string> words2{ "Long", "live", "the", "king." }; ❷
  swap_ranges(words1.begin(), words1.end(), words2.begin()); ❸
  REQUIRE(words1 == vector<string>{ "Long", "live", "the", "king." }); ❹
  REQUIRE(words2 == vector<string>{ "The", "king", "is", "dead." }); ❺
}
```

После создания двух векторов, содержащих строки ❶ ❷, вызывается `swap` с `words1` и `words2` как последовательностями для замены ❸. В результате `words1` и `words2` меняют содержимое ❹ ❺.

transform

Алгоритм `transform` изменяет элементы одной последовательности и записывает их в другую.

Алгоритм вызывает `unary_op` для каждого элемента целевой последовательности и выводит его в выходную последовательность или вызывает `binary_op` для соответствующих элементов каждой целевой последовательности.

```
OutputIterator transform([ep], ipt_begin1, ipt_end1, result, unary_op);
OutputIterator transform([ep], ipt_begin1, ipt_end1, ipt_begin2,
                         result, binary_op);
```

Аргументы

- Необязательная политика выполнения `std::execution`, `ep` (по умолчанию: `std::execution::seq`).
- Пара объектов `InputIterator`, `ipt_begin1` и `ipt_end1`, представляющих целевую последовательность.
- Дополнительный `InputIterator`, `ipt_begin2`, представляющий вторую целевую последовательность. Нужно убедиться, что эта вторая целевая последовательность имеет как минимум столько же элементов, сколько и первая целевая последовательность.
- `OutputIterator`, результат, представляющий начало выходной последовательности.

- Унарная операция, `unary_op`, которая преобразует элементы целевой последовательности в элементы выходной последовательности. Если предоставить две целевые последовательности, вместо `unary_op` предоставляется бинарная операция `binary_op`, которая принимает элемент из каждой целевой последовательности и преобразует каждую в элемент выходной последовательности.

Сложность

Линейная. Алгоритм вызывает `unary_op` или `binary_op` ровно `distance(ipt _begin1, ipt_end1)` раз.

Примеры

```
#include <algorithm>
#include <boost/algorithm/string/case_conv.hpp>

TEST_CASE("transform") {
  vector<string> words1{ "farewell", "hello", "farewell", "hello" }; ❶
  vector<string> result1;
  auto upper = [](string x) { ❷
    boost::algorithm::to_upper(x);
    return x;
  };
  transform(words1.begin(), words1.end(), back_inserter(result1), upper); ❸
  REQUIRE(result1 == vector<string>{ "FAREWELL", "HELLO",
                                     "FAREWELL", "HELLO" }); ❹

  vector<string> words2{ "light", "human", "bro", "quantum" }; ❺
  vector<string> words3{ "radar", "robot", "pony", "bit" }; ❻
  vector<string> result2;
  auto portmantize = [](const auto &x, const auto &y) { ❼
    const auto x_letters = min(size_t{ 2 }, x.size());
    string result{ x.begin(), x.begin() + x_letters };
    const auto y_letters = min(size_t{ 3 }, y.size());
    result.insert(result.end(), y.end() - y_letters, y.end() );
    return result;
  };
  transform(words2.begin(), words2.end(), words3.begin(),
            back_inserter(result2), portmantize); ❽
  REQUIRE(result2 == vector<string>{ "lidar", "hubot", "brony", "qubit" }); ❾
}
```

После создания вектора, содержащего строку ❶, создается лямбда-предикат под названием `upper`, который принимает `string` по значению и преобразует ее в верхний регистр, используя алгоритм Boost `to_upper`, описанный в главе 15 ❷. `transform` вызывается с `words1` в качестве целевой последовательности, `back_inserter` в качестве пустого вектора `results1` и `upper` в качестве унарной операции ❸. После преобразования `result1` содержит версию words1 ❹ в верхнем регистре.

Во втором примере создаются два вектора, содержащие строки ❺ ❻. Также создается лямбда-предикат под названием `portmantize`, который принимает два объекта

строки ❼. Лямбда возвращает новую строку, содержащую до двух букв от начала первого аргумента и до трех букв от конца второго аргумента. Две целевые последовательности, `back_inserter`, передаются в пустой вектор `results2` и вызывается `portmantize` ❽. `result2` содержит комбинацию содержимого `words1` и `words2` ❾.

replace

Алгоритм `replace` заменяет некоторые элементы последовательности новым элементом.

Алгоритм ищет элементы целевой последовательности **`x`**, для которых **`x`** `== old_ref` или `pred(x) == true`, и присваивает их `new_ref`.

```
void replace([ep], fwd_begin, fwd_end, old_ref, new_ref);
void replace_if([ep], fwd_begin, fwd_end, pred, new_ref);
void replace_copy([ep], fwd_begin, fwd_end, result, old_ref, new_ref);
void replace_copy_if([ep], fwd_begin, fwd_end, result, pred, new_ref);
```

Аргументы

- Необязательная политика выполнения `std::execution`, `ep` (по умолчанию: `std::execution::seq`).
- Пара `ForwardIterators`, `fwd_begin` и `fwd_end`, представляющая целевую последовательность.
- `OutputIterator`, результат, представляющий начало выходной последовательности.
- `const` ссылка `old_ref`, представляющая элемент для поиска.
- Унарный предикат `pred`, определяющий, соответствует ли элемент критериям замены.
- `const`-ссылка `new_ref`, представляющая элемент для замены.

Сложность

Линейная. Алгоритм вызывает `pred` ровно `distance(fwd_begin, fwd_end)` раз.

Дополнительные требования

Элементы, содержащиеся в каждой последовательности, должны быть сопоставимы с `old_ref` и присваиваемы `new_ref`.

Сначала вводится пространство имен `std::literals` ❶, чтобы позже можно было использовать литерал `string_view`. После создания вектора, содержащего строковые объекты ❷, вызывается `replace`с `vector` ❸, чтобы заменить все экземпляры `try` на `spoon` ❹.

Примеры

```
#include <algorithm>
#include <string_view>

TEST_CASE("replace") {
  using namespace std::literals; ❶
  vector<string> words1{ "There", "is", "no", "try" }; ❷
  replace(words1.begin(), words1.end(), "try"sv, "spoon"sv); ❸
  REQUIRE(words1 == vector<string>{ "There", "is", "no", "spoon" }); ❹

  const vector<string> words2{ "There", "is", "no", "spoon" }; ❺
  vector<string> words3{ "There", "is", "no", "spoon" }; ❻
  auto has_two_os = [](const auto& x) { ❼
    return count(x.begin(), x.end(), 'o') == 2;
  };
  replace_copy_if(words2.begin(), words2.end(), words3.begin(), ❽
                  has_two_os, "try"sv);
  REQUIRE(words3 == vector<string>{ "There", "is", "no", "try" }); ❾
}
```

Во втором примере создаются два вектора, содержащие строку ❺ ❻ и лямбда-предикат с названием `has_two_os`, который принимает строку и возвращает `true`, если она содержит ровно два `os` ❼. Затем `words2` передается в качестве целевой последовательности и `words3` в качестве последовательности назначения в `replace_copy_if`, который применяет `has_two_os` к каждому элементу `words2` и заменяет элементы, имеющие значение `true`, на `try` ❽. Результатом является то, что `words2` не затрагивается, а в `words3` элемент `spoon` заменяется на `try` ❾.

fill

Алгоритм `fill` заполняет последовательность некоторым значением.

Алгоритм записывает значение в каждый элемент целевой последовательности. Функция `fill_n` возвращает `opt_begin+n`.

```
void fill([ep], fwd_begin, fwd_end, value);
OutputIterator fill_n([ep], opt_begin, n, value);
```

Аргументы

- Необязательная политика выполнения `std::execution`, `ep` (по умолчанию: `std::execution::seq`).
- `ForwardIterator`, `fwd_begin`, представляющий начало целевой последовательности.
- `ForwardIterator`, `fwd_end`, представляющий элемент за последним элементом последовательности.
- `Size n`, представляющий количество элементов.
- `Value` для записи в каждый элемент целевой последовательности.

Сложность

Линейная. Алгоритм присваивает `value` ровно `distance(fwd_begin, fwd_end)` или `n` раз.

Дополнительные требования

- Параметр `value` должен быть записываемым в последовательность.
- Объекты типа `Size` должны иметь возможность преобразования в целочисленный тип.

Примеры

```
#include <algorithm>

// Если проверяющие проверяющих проверяют проверяющих,
// кто проверяет проверяющих проверяющих?
TEST_CASE("fill") {
  vector<string> answer1(6); ❶
  fill(answer1.begin(), answer1.end(), "police"); ❷
  REQUIRE(answer1 == vector<string>{ "police", "police", "police",
                                     "police", "police", "police" }); ❸

  vector<string> answer2; ❹
  fill_n(back_inserter(answer2), 6, "police"); ❺
  REQUIRE(answer2 == vector<string>{ "police", "police", "police",
                                     "police", "police", "police" }); ❻
}
```

Сначала инициализируется вектор, содержащий строку с шестью пустыми элементами ❶. Затем вызывается `fill`, используя этот вектор в качестве целевой последовательности, и `police` в качестве значения ❷. В результате вектор содержит шесть `police` ❸.

Во втором примере инициализируется пустой вектор, содержащий объекты строки ❹. Затем вызывается `fill_n` с указателем `back_inserter`, указывающим на пустой вектор длиной 6 и `police` в качестве значения ❺. Результат тот же, что и раньше: `vector` содержит шесть `police` ❻.

generate

Алгоритм `generate` заполняет последовательность, вызывая функциональный объект.

Алгоритм вызывает `generator` и присваивает результат целевой последовательности. Функция `generate_n` возвращает `opt_begin+n`.

```
void generate([ep], fwd_begin, fwd_end, generator);
OutputIterator generate_n([ep], opt_begin, n, generator);
```

Аргументы

- Необязательная политика выполнения `std::execution`, `ep` (по умолчанию: `std::execution::seq`).
- `ForwardIterator`, `fwd_begin`, представляющий начало целевой последовательности.
- `ForwardIterator`, `fwd_end`, представляющий элемент, следующий за последним элементом последовательности.
- `Size n` представляет количество элементов.
- `generator`, который при вызове без аргументов создает элемент для записи в целевую последовательность.

Сложность

Линейная. Алгоритм вызывает `generator` ровно `distance(fwd_begin, fwd_end)` или `n` раз.

Дополнительные требования

- Параметр `value` должен быть записываемым в последовательность.
- Объекты типа `Size` должны иметь возможность преобразования в целочисленный тип.

Примеры

```
#include <algorithm>

TEST_CASE("generate") {
  auto i{ 1 }; ❶
  auto pow_of_2 = [&i]() { ❷
    const auto tmp = i;
    i *= 2;
    return tmp;
  };
  vector<int> series1(6); ❸
  generate(series1.begin(), series1.end(), pow_of_2); ❹
  REQUIRE(series1 == vector<int>{ 1, 2, 4, 8, 16, 32 }); ❺

  vector<int> series2; ❻
  generate_n(back_inserter(series2), 6, pow_of_2); ❼
  REQUIRE(series2 == vector<int>{ 64, 128, 256, 512, 1024, 2048 }); ❽
}
```

Сначала инициализируется `int` под названием `i` со значением 1 ❶. Затем создается лямбда-предикат с именем `pow_of_2`, который принимает `i` по ссылке ❷. Каждый раз при вызове `pow_of_2` он удваивает i и возвращает значение непосредственно перед удвоением. Затем инициализируется `vector` объектов `int` шестью элементами ❸. Затем вызывается `generate` с `vector` в качестве целевой последовательности

и pow_of_2 в качестве генератора ❹. В результате vector содержит первые шесть степеней двойки ❺.

Во втором примере инициализируется пустой вектор объектов int ❻. Затем вызывается generate_n, используя back_inserter для вашего пустого vector размером 6, и pow_of_2 в качестве генератора ❼. Результатом являются следующие шесть степеней двойки ❽. Обратите внимание, что pow_of_2 имеет состояние, потому что он захватывает i по ссылке.

remove

Алгоритм remove удаляет определенные элементы из последовательности.

Алгоритм перемещает все элементы, где pred вычисляется как true или где элемент равен value, таким образом, что порядок остальных элементов сохраняется, и возвращает итератор, указывающий на первый перемещенный элемент. Этот итератор называется логическим окончанием полученной последовательности. Физический размер последовательности остается неизменным, и за вызовом удаления обычно следует вызов метода erase контейнера.

```
ForwardIterator remove([ep], fwd_begin, fwd_end, value);
ForwardIterator remove_if([ep], fwd_begin, fwd_end, pred);
ForwardIterator remove_copy([ep], fwd_begin, fwd_end, result, value);
ForwardIterator remove_copy_if([ep], fwd_begin, fwd_end, result, pred);
```

Аргументы

- Необязательная политика выполнения std::execution, ep (по умолчанию: std::execution::seq).
- Пара ForwardIterators, fwd_begin и fwd_end, представляющая целевую последовательность.
- OutputIterator, результат, представляющий последовательность назначения (при копировании).
- value, представляющий элемент для удаления.
- Унарный предикат pred, определяющий, соответствует ли элемент критериям удаления.

Сложность

Линейная. Алгоритм вызывает pred или сравнивается с value ровно distance(fwd_begin, fwd_end) раз.

Дополнительные требования

- Элементы целевой последовательности должны быть перемещаемыми.
- При копировании элементы должны быть копируемыми, а целевая последовательность и последовательность назначения не должны перекрываться.

Примеры

```
#include <algorithm>

TEST_CASE("remove") {
  auto is_vowel = [](char x) { ❶
    const static string vowels{ "aeiouAEIOU" };
    return vowels.find(x) != string::npos;
  };
  string pilgrim = "Among the things Billy Pilgrim could not change "
                   "were the past, the present, and the future."; ❷
  const auto new_end = remove_if(pilgrim.begin(), pilgrim.end(), is_vowel); ❸

  REQUIRE(pilgrim == "mng th thngs Blly Plgrm cld nt chng wr th pst, "
                     "th prsnt, nd th ftr.present, and the future."); ❹

  pilgrim.erase(new_end, pilgrim.end()); ❺
  REQUIRE(pilgrim == "mng th thngs Blly Plgrm cld nt chng wr th "
                     "pst, th prsnt, nd th ftr."); ❻
}
```

Сначала создается лямбда-выражение `is_vowel`, которое возвращает `true`, если данный символ является гласным ❶. Затем создается строка с именем `pilgrim`, содержащая предложение ❷. Затем вызывается `remove_if` с `pilgrim` в качестве целевого предложения и `is_vowel` в качестве предиката ❸. Это устраняет все гласные в предложении, сдвигая оставшиеся символы влево каждый раз, когда `remove_if` встречает гласную. В результате `pilgrim` содержит оригинальное предложение с удаленными гласными, а также фразу `present, and the future.` ❹. Эта фраза содержит 24 символа, что в точности соответствует количеству гласных, которые были удалены из оригинального предложения. Фраза `present, and the future` — это сдвиг оставшейся строки во время удаления.

Чтобы устранить эти остатки, сохраняется итератор `new_end`, который возвращает `remove_if`. Он указывает на элемент, следующий за последним символом в новой целевой последовательности, `p` в `present, and the future`. Чтобы устранить это, используется метод `erase` в `pilgrim`, который имеет перегрузку, принимающую полуоткрытый диапазон. В нее передается логический конец, возвращаемый `remove_if`, `new_end` в качестве начального итератора. Также передается `pilgrim.end()` в качестве конечного итератора ❺. В результате `pilgrim` теперь равен исходному предложению с удаленными гласными ❻.

Эта комбинация удаления (или `remove_if`) и метода стирания, которая называется *идиомой стирания-удаления*, используется повсеместно.

unique

Алгоритм `unique` удаляет лишние элементы из последовательности.

Алгоритм перемещает все повторяющиеся элементы, где `pred` вычисляется как `true` или где элементы равны, так что оставшиеся элементы уникальны по отношению к своим соседям, и первоначальный порядок сохраняется. Возвращается итератор,

указывающий на новый логический конец. Как и в случае с `std::remove`, физическое хранилище не меняется.

```
ForwardIterator unique([ep], fwd_begin, fwd_end, [pred]);
ForwardIterator unique_copy([ep], fwd_begin, fwd_end, result, [pred]);
```

Аргументы

- Необязательная политика выполнения `std::execution`, `ep` (по умолчанию: `std::execution::seq`).
- Пара `ForwardIterators`, `fwd_begin` и `fwd_end`, представляющая целевую последовательность.
- `OutputIterator`, результат, представляющий последовательность назначения (при копировании).
- Двоичный предикат `pred`, который определяет, равны ли два элемента.

Сложность

Линейная. Алгоритм вызывает `pred` ровно `distance (fwd_begin, fwd_end) – 1` раз.

Дополнительные требования

- Элементы целевой последовательности должны быть переносимыми.
- При копировании элементы целевой последовательности должны копироваться, а диапазоны целевой последовательности и назначения не должны перекрываться.

Пример

```
#include <algorithm>

TEST_CASE("unique") {
  string without_walls = "Wallless"; ❶
  const auto new_end = unique(without_walls.begin(), without_walls.end()); ❷
  without_walls.erase(new_end, without_walls.end()); ❸
  REQUIRE(without_walls == "Wales"); ❹
}
```

Сначала создается `string`, содержащая слово с несколькими повторяющимися символами ❶. Затем вызывается `unique` с `string` в качестве целевой последовательности ❷. Это возвращает логический конец, который присваивается `new_end`. Далее стирается диапазон, начинающийся с `new_end` и заканчивающийся `without_walls.end()` ❸. Это следствие идиомы стирания-удаления: остается содержимое `Wales`, в котором последовательно располагаются только уникальные символы ❹.

reverse

Алгоритм `reverse` меняет порядок последовательности.

Алгоритм обращает последовательность, меняя ее элементы или копируя их в целевую последовательность.

```
void reverse([ep], bi_begin, bi_end);
OutputIterator reverse_copy([ep], bi_begin, bi_end, result);
```

Аргументы

- Необязательная политика выполнения `std::execution`, `ep` (по умолчанию: `std::execution::seq`).
- Пара `BidirectionalIterators`, `bi_begin` и `bi_end`, представляющая целевую последовательность.
- `OutputIterator`, результат, представляющий последовательность назначения (при копировании).

Сложность

Линейная. Алгоритм вызывает `swap` ровно `distance(bi_begin, bi_end)/2` раз.

Дополнительные требования

- Элементы целевой последовательности должны быть заменяемыми.
- При копировании элементы целевой последовательности должны быть копируемыми, а диапазоны целевой последовательности и назначения не должны перекрываться.

Пример

```
#include <algorithm>

TEST_CASE("reverse") {
  string stinky = "diaper"; ❶
  reverse(stinky.begin(), stinky.end()); ❷
  REQUIRE(stinky == "repaid"); ❸
}
```

Сначала создается строка, содержащая слово `diaper` ❶. Далее вызывается `reverse` с этой строкой в качестве целевой последовательности ❷. Результатом является слово `repaid` ❸.

sample

Алгоритм `sample` генерирует случайные стабильные подпоследовательности.

Алгоритм выбирает `min(pop_end - pop_begin, n)` элементов из последовательности совокупности. Что несколько неинтуитивно, образец будет отсортирован тогда и только тогда, когда `ipt_begin` является прямым итератором. Возвращает окончание последовательности назначения.

```
OutputIterator sample([ep], ipt_begin, ipt_end, result, n, urb_generator);
```

Аргументы

- Необязательная политика выполнения `std::execution`, `ep` (по умолчанию: `std::execution::seq`).
- Пара объектов `InputIterator`, `ipt_begin` и `ipt_end`, представляющих последовательность совокупности (последовательность для выборки).
- `OutputIterator`, результат, представляющий последовательность назначения.
- `Distance n`, представляющий количество элементов для выборки.
- `UniformRandomBitGeneratorurb_generator`, такой как вихрь Мерсенна `std::mt19937_64`, представленный в главе 12.

Сложность

Линейная. Сложность алгоритма зависит от `distance(ipt_begin, ipt_end)`.

Пример

```
#include <algorithm>
#include <map>
#include <string>
#include <iostream>
#include <iomanip>
#include <random>

using namespace std;

const string population = "ABCD"; ❶
const size_t n_samples{ 1'000'000 }; ❷
mt19937_64 urbg; ❸

void sample_length(size_t n) { ❹
  cout << "-- Length " << n << " --\n";
  map<string, size_t> counts; ❺
  for (size_t i{}; i < n_samples; i++) {
    string result;
    sample(population.begin(), population.end(),
           back_inserter(result), n, urbg); ❻
    counts[result]++;
  }
  for (const auto[sample, n] : counts) { ❼
    const auto percentage = 100 * n / static_cast<double>(n_samples);
    cout << percentage << " '" << sample << "'\n"; ❽
  }
}

int main() {
  cout << fixed << setprecision(1); ❾
  sample_length(0); ❿
```

```
  sample_length(1);
  sample_length(2);
  sample_length(3);
  sample_length(4);
}
-------------------------------------------------------------------------------
-- Length 0 --
100.0 ''
-- Length 1 --
25.1 'A'
25.0 'B'
25.0 'C'
24.9 'D'
-- Length 2 --
16.7 'AB'
16.7 'AC'
16.6 'AD'
16.6 'BC'
16.7 'BD'
16.7 'CD'
-- Length 3 --
25.0 'ABC'
25.0 'ABD'
25.0 'ACD'
25.0 'BCD'
-- Length 4 --
100.0 'ABCD'
```

Сначала создается `const string` под названием `population`, содержащая буквы ABCD ❶. Также инициализируется `const size_t` под названием `n_samples`, равным миллиону ❷, и вихрь Мерсенна с именем `urbg` ❸. Все эти объекты имеют статическую длительность хранения.

Кроме того, инициализируется функция `sample_length`, которая принимает один аргумент `size_t` с названием `n` ❹. Внутри функции создается `map` с типами `string` для объектов `size_t` ❺, который будет подсчитывать частоту каждого вызова образца. Внутри цикла `for` вызывается `sample` с `population` в качестве последовательности заполнения, `back_inserter` в строке `result` в качестве последовательности назначения, `n` в качестве длины выборки и `urbg` в качестве генератора случайных битов ❻.

После миллиона итераций перебирается каждый элемент `counts` ❼ и выводится распределение вероятностей каждой выборки для заданной длины `n` ❽.

В `main` настраивается форматирование с плавающей точкой с помощью `fixed` и `setprecision` ❾. Наконец, вызывается `sample_length` с каждым значением от 0 до 4 включительно ❿.

Поскольку `string` предоставляет итераторы произвольного доступа, `sample` предоставляет стабильные (отсортированные) выборки.

ПРЕДУПРЕЖДЕНИЕ

Обратите внимание, что выходные данные не содержат несортированных выборок, таких как DC или CAB. Такое поведение сортировки не всегда очевидно из названия алгоритма, поэтому будьте осторожны!

shuffle

Алгоритм `shuffle` генерирует случайные перестановки.

Алгоритм случайным образом меняет целевую последовательность так, что каждая возможная перестановка этих элементов имеет равную вероятность появления.

```
void shuffle(rnd_begin, rnd_end, urb_generator);
```

Аргументы

- Пара `RandomAccessIterators`, `rnd_begin` и `rnd_end`, представляющая целевую последовательность.
- `UniformRandomBitGenerator urb_generator`, такой как вихрь Мерсенна `std::mt19937_64`, представленный в главе 12.

Сложность

Линейная. Алгоритм меняется ровно `distance(rnd_begin, rnd_end) –1` раз.

Дополнительные требования

Элементы целевой последовательности должны быть заменяемыми.

Пример

```
#include <algorithm>
#include <map>
#include <string>
#include <iostream>
#include <random>
#include <iomanip>

using namespace std;

int main() {
  const string population = "ABCD"; ❶
  const size_t n_samples{ 1'000'000 }; ❷
  mt19937_64 urbg; ❸
  map<string, size_t> samples; ❹
  cout << fixed << setprecision(1); ❺
  for (size_t i{}; i < n_samples; i++) {
    string result{ population }; ❻
    shuffle(result.begin(), result.end(), urbg); ❼
```

```
    samples[result]++; ❽
  }
  for (const auto[sample, n] : samples) { ❾
    const auto percentage = 100 * n / static_cast<double>(n_samples);
    cout << percentage << " '" << sample << "'\n"; ❿
  }
}
```

```
4.2 'ABCD'
4.2 'ABDC'
4.1 'ACBD'
4.2 'ACDB'
4.2 'ADBC'
4.2 'ADCB'
4.2 'BACD'
4.2 'BADC'
4.1 'BCAD'
4.2 'BCDA'
4.1 'BDAC'
4.2 'BDCA'
4.2 'CABD'
4.2 'CADB'
4.1 'CBAD'
4.1 'CBDA'
4.2 'CDAB'
4.1 'CDBA'
4.2 'DABC'
4.2 'DACB'
4.2 'DBAC'
4.1 'DBCA'
4.2 'DCAB'
4.2 'DCBA'
```

Сначала создается `const string` под названием `population`, содержащая буквы ABCD ❶. Также инициализируется `const size_t` под названием `n_samples`, равный миллиону ❷, вихрь Мерсенна под названием `urbg` ❸ и `map` с типами `string` для объектов `size_t` ❹, который будет подсчитывать частоту каждой выборки в случайном порядке. Кроме того, настраивается форматирование с плавающей запятой с помощью `fixed` и `setprecision` ❺.

Внутри цикла `for` совокупность копируется в новую строку под названием `sample`, потому что `shuffle` изменяет целевую последовательность ❻. Затем вызывается `shuffle` с `result` в качестве целевой последовательности и `urbg` в качестве генератора случайных битов ❼ и записывается результат в `samples` ❽.

Наконец, каждый элемент в `samples` перебирается ❾ и выводится распределение вероятностей каждой выборки ❿.

Обратите внимание, что, в отличие от `sample`, `shuffle` всегда создает *неупорядоченное* распределение элементов.

Сортировка и связанные операции

Операция сортировки — это алгоритм, который переупорядочивает последовательность некоторым желаемым способом.

Каждый алгоритм сортировки имеет две версии: одна, которая принимает функциональный объект, называемый *оператором сравнения*, и другая, которая использует `operator<`. Оператор сравнения — это функциональный объект, который вызывается для сравнения двух объектов. Он возвращает `true`, если первый аргумент *меньше* второго аргумента; в противном случае возвращается `false`. Сортировка интерпретации `x<y` состоит в том, что `x` сортируется перед `y`. Все алгоритмы, описанные в этом разделе, находятся в заголовке `<algorithm>`.

ПРИМЕЧАНИЕ

Обратите внимание, что `operator<` является допустимым оператором сравнения.

Операторы сравнения должны быть транзитивными. Это означает, что для любых элементов `a`, `b` и `c` оператор сравнения `comp` должен сохранять следующее соотношение: если `comp(a, b)` и `comp(b, c)`, то `comp(a, c)`. Это должно иметь смысл: если `a` упорядочено до `b`, а `b` упорядочено до `c`, то `a` должно быть упорядочено до `c`.

sort

Алгоритм `sort` сортирует последовательность (нестабильно).

ПРИМЕЧАНИЕ

Стабильная сортировка сохраняет относительный порядок предварительной сортировки равных элементов, тогда как нестабильная может изменить их порядок.

Алгоритм сортирует целевую последовательность на месте.

```
void sort([ep], rnd_begin, rnd_end, [comp]);
```

Аргументы

- Необязательная политика выполнения `std::execution`, `ep` (по умолчанию: `std::execution::seq`).
- Пара `RandomAccessIterators`, `rnd_begin` и `rnd_end`, представляющая целевую последовательность.
- Необязательный оператор сравнения `comp`.

Сложность

Квазилинейная. `O(N log N)`, где `N = distance(rnd_begin, rnd_end)`

Дополнительные требования

Элементы целевой последовательности должны быть заменяемыми, перемещаемыми и присваиваемыми.

Пример

```
#include <algorithm>

TEST_CASE("sort") {
  string goat_grass{ "spoilage" }; ❶
  sort(goat_grass.begin(), goat_grass.end()); ❷
  REQUIRE(goat_grass == "aegilops");❸
}
```

Сначала создается `string`, содержащая слово `spoilage` ❶. Затем вызывается `sort` с этой строкой в качестве целевой последовательности ❷. В результате `goat_grass` теперь содержит слово `aegilops` (род быстро распространяющихся сорняков) ❸.

stable_sort

Алгоритм `stable_sort` сортирует последовательность стабильно.

Алгоритм сортирует целевую последовательность на месте. Равные элементы сохраняют свой первоначальный порядок.

```
void stable_sort([ep], rnd_begin, rnd_end, [comp]);
```

Аргументы

- Необязательная политика выполнения `std::execution`, `ep` (по умолчанию: `std::execution::seq`).
- Пара `RandomAccessIterators`, `rnd_begin` и `rnd_end`, представляющая целевую последовательность.
- Необязательный оператор сравнения `comp`.

Сложность

Полилог-линейная. $O(N \log^2 N)$, где $N = distance(rnd\_begin, rnd\_end)$. Если доступна дополнительная память, сложность уменьшается до квазилинейной.

Дополнительные требования

Элементы целевой последовательности должны быть заменяемыми, перемещаемыми и присваиваемыми.

В этом примере `string` сортируется с использованием *надстрочных* и *подстрочных символов*. В типографии надстрочный символ — это буква, часть которой простирается над так называемой средней линией шрифта. Подстрочный символ — это буква,

часть которой располагается ниже так называемой базовой линии. Буквы, обычно набираемые подстрочно: *g, j, p, q* и *y*. Буквы, обычно набираемые надстрочно: *b, d, f, h, k, l* и *t*. В этом примере выполняется поиск `stable_sort`, чтобы все буквы с надстрочными символами отображались до всех остальных букв, а буквы с подстрочными — после всех остальных. Буквы без надстрочных и подстрочных символов располагаются посередине. В `stable_sort` относительный порядок букв с общей категоризацией по надстрочным/подстрочным символам не должен изменяться.

Пример

```
#include <algorithm>

enum class CharCategory { ❶
  Ascender,
  Normal,
  Descender
};

CharCategory categorize(char x) { ❷
  switch (x) {
    case 'g':
    case 'j':
    case 'p':
    case 'q':
    case 'y':
      return CharCategory::Descender;
    case 'b':
    case 'd':
    case 'f':
    case 'h':
    case 'k':
    case 'l':
    case 't':
      return CharCategory::Ascender;
  }
  return CharCategory::Normal;
}

bool ascension_compare(char x, char y) { ❸
  return categorize(x) < categorize(y);
}

TEST_CASE("stable_sort") {
  string word{ "outgrin" }; ❹
  stable_sort(word.begin(), word.end(), ascension_compare); ❺
  REQUIRE(word == "touring"); ❻
}
```

Сначала определяется класс `enum` под названием `CharCategory`, который принимает три возможных значения: `Ascender`, `Normal` или `Descender` ❶. Затем определяется

функция, которая классифицирует данный символ в `CharCategory` ❷. (Вспомните из раздела «Операторы Switch» на с. 296, что метки «проваливаются», если не добавить `break`.) Также определяется функция `ascension_compare`, которая преобразует два заданных объекта `char` в объекты `CharCategory` и сравнивает их с помощью `operator<` ❸. Поскольку объекты класса `enum` неявно преобразуются в объекты `int` и поскольку `CharCategory` определяется со значениями в предполагаемом порядке, буквы с надстрочными символами будут отсортированы перед обычными буквами, а те — перед буквами с подстрочными символами.

В тестовом примере инициализируется строка, содержащая слово `outgrin` ❹. Затем вызывается `stable_sort` с этой строкой в качестве целевой последовательности и `ascension_compare` в качестве оператора сравнения ❺. Результатом является то, что `word` теперь содержит `touring` ❻. Обратите внимание, что `t`, единственная буква с надстрочным символом, появляется перед всеми обычными символами (которые находятся в том же порядке, что и в `outgrin`), которые в свою очередь расположены перед `g`, единственным потомком.

partial_sort

Алгоритм `partial_sort` сортирует последовательность по двум группам.

При изменении алгоритм сортирует первые `(rnd_middle - rnd_begin)` элементы в целевой последовательности, поэтому все элементы в `rnd_begin`до`rnd_middle` меньше, чем остальные элементы. При копировании алгоритм помещает первые отсортированные элементы `min(distance(ipt_begin, ipt_end), distance(rnd_begin, rnd_end))` в целевую последовательность и возвращает итератор, указывающий на конец целевой последовательности.

По сути, частичная сортировка позволяет найти первые несколько элементов отсортированной последовательности без необходимости сортировки всей последовательности. Например, если бы была последовательность D C B A, можно было бы частично отсортировать первые два элемента и получить результат A B D C. Первые два элемента будут такими же, как если бы вы отсортировали всю последовательность, но остальные элементы не изменятся.

```
void partial_sort([ep], rnd_begin, rnd_middle, rnd_end, [comp]);
RandomAccessIterator partial_sort_copy([ep], ipt_begin, ipt_end,
                                       rnd_begin, rnd_end, [comp]);
```

Аргументы

- Необязательная политика выполнения `std::execution`, `ep` (по умолчанию: `std::execution::seq`).
- При изменении — трио `RandomAccessIterators`, `rnd_begin`, `rnd_middle` и `rnd_end`, представляющих целевую последовательность.

- При копировании — пара `ipt_begin` и `ipt_end`, представляющая целевую последовательность, и пара `rnd_begin` и `rnd_end`, представляющая последовательность назначения.
- Необязательный оператор сравнения `comp`.

Сложность

Квазилинейная. `O(N log N)`, где `N = distance(rnd_begin, rnd_end) * log(distance (rnd_begin, rnd_middle)` или `distance(rnd_begin, rnd_end) * log(min(distance(rnd_ begin, rnd_end), distance(ipt_begin, ipt_end))` для варианта с копированием.

Дополнительные требования

Элементы целевой последовательности должны быть заменяемыми, перемещаемыми и присваиваемыми.

Примеры

```
#include <algorithm>

bool ascension_compare(char x, char y) {
--пропуск--
}

TEST_CASE("partial_sort") {
  string word1{ "nectarous" }; ❶
  partial_sort(word1.begin(), word1.begin() + 4, word1.end()); ❷
  REQUIRE(word1 == "acentrous"); ❸

  string word2{ "pretanning" }; ❹
  partial_sort(word2.begin(), word2.begin() + 3, ❺
               word2.end(), ascension_compare);
  REQUIRE(word2 == "trepanning"); ❻
}
```

Сначала инициализируется строка, содержащая слово `nectarous` ❶. Затем вызывается `part_sort` с этой строкой в качестве целевой последовательности и пятой буквой `(a)` в качестве второго аргументом для `component_sort` ❷. В результате последовательность теперь содержит слово `acentrous` ❸. Обратите внимание, что первые четыре буквы `acentrous` отсортированы и что они меньше, чем остальные символы в последовательности.

Во втором примере инициализируется строка, содержащая слово `pretanning` ❹, которое используется в качестве целевой последовательности для `partial_sort` ❺. В этом примере указывается четвертый символ `(t)` в качестве второго аргумента для `component_sort` и используется функция `ascension_compare` из примера `stable_sort` в качестве оператора сравнения. В результате последовательность теперь содержит слово `trepanning` ❻. Обратите внимание, что первые три буквы отсортированы

в соответствии с `ascension_compare` и ни один из оставшихся символов во втором аргументе `part_sort` z не меньше, чем первые три символа.

ПРИМЕЧАНИЕ

Технически операторы REQUIRE в предыдущем примере могут не работать в некоторых реализациях стандартной библиотеки. Поскольку std::partial_sort не гарантируется как стабильная, результаты могут отличаться.

is_sorted

Алгоритм `is_sorted` определяет, отсортирована ли последовательность.

Алгоритм возвращает `true`, если целевая последовательность отсортирована в соответствии с `operator<` или `comp`, если он задан. Алгоритм `is_sorted_until` возвращает итератор, указывающий на первый несортированный элемент, или `rnd_end`, если целевая последовательность отсортирована.

```
bool is_sorted([ep], rnd_begin, rnd_end, [comp]);
ForwardIterator is_sorted_until([ep], rnd_begin, rnd_end, [comp]);
```

Аргументы

- Необязательная политика выполнения `std::execution`, `ep` (по умолчанию: `std::execution::seq`).
- Пара `RandomAccessIterators`, `rnd_begin` и `rnd_end`, представляющая целевую последовательность.
- Необязательный оператор сравнения `comp`.

Сложность

Линейная. Алгоритм выполняет сравнение `distance(rnd_begin, rnd_end)` раз.

Примеры

```
#include <algorithm>

bool ascension_compare(char x, char y) {
--пропуск--
}

TEST_CASE("is_sorted") {
  string word1{ "billowy" }; ❶
  REQUIRE(is_sorted(word1.begin(), word1.end())); ❷

  string word2{ "floppy" }; ❸
  REQUIRE(word2.end() == is_sorted_until(word2.begin(), ❹
                                         word2.end(), ascension_compare));
}
```

Сначала создается `string`, содержащая слово `billowy` ❶. Затем вызывается функция `is_sorted` с этой `string` в качестве целевой последовательности, которая возвращает `true` ❷.

Во втором примере создается `string`, содержащая слово `floppy` ❸. Затем вызывается `is_sorted_until` с этой строкой в качестве целевой последовательности, которая возвращает `rnd_end`, поскольку последовательность отсортирована ❹.

nth_element

Алгоритм `nth_element` помещает определенный элемент в последовательности в правильную отсортированную позицию.

Этот алгоритм частичной сортировки изменяет целевую последовательность следующим образом: элемент в позиции, на которую указывает `rnd_nth`, находится в той позиции, как если бы весь диапазон был отсортирован. Все элементы от `rnd_begin` до `rnd_nth-1` будут меньше, чем `rnd_nth`. Если `rnd_nth == rnd_end`, функция не выполняет никаких действий.

```
bool nth_element([ep], rnd_begin, rnd_nth, rnd_end, [comp]);
```

Аргументы

- Необязательная политика выполнения `std::execution`, `ep` (по умолчанию: `std::execution::seq`).
- Трио из `RandomAccessIterators`, `rnd_begin`, `rnd_nth` и `rnd_end`, представляющее целевую последовательность.
- Необязательный оператор сравнения `comp`.

Сложность

Линейная. Алгоритм выполняет сравнение `distance(rnd_begin, rnd_end)` раз.

Дополнительные требования

Элементы целевой последовательности должны быть заменяемыми, перемещаемыми и присваиваемыми.

Пример

```
#include <algorithm>

TEST_CASE("nth_element") {
  vector<int> numbers{ 1, 9, 2, 8, 3, 7, 4, 6, 5 }; ❶
  nth_element(numbers.begin(), numbers.begin() + 5, numbers.end()); ❷
  auto less_than_6th_elem = [&elem=numbers[5]](int x) { ❸
    return x < elem;
  };
  REQUIRE(all_of(numbers.begin(), numbers.begin() + 5, less_than_6th_elem)); ❹
  REQUIRE(numbers[5] == 6 ); ❺
}
```

Сначала создается `vector` объектов `int`, содержащий числовую последовательность от 1 до 10 включительно ❶. Затем вызывается `nth_element` с этим `vector` в качестве целевой последовательности ❷. Затем инициализируется лямбда-предикат под названием `less_than_6th_elem`, который сравнивает `int` с шестым элементом `numbers` с помощью `operator<` ❸. Это позволяет проверить, что все элементы перед шестым элементом меньше шестого элемента ❹. Шестой элемент равен 6 ❺.

Бинарный поиск

Алгоритмы бинарного поиска предполагают, что целевая последовательность уже отсортирована.

Эти алгоритмы имеют желательные характеристики сложности по сравнению с общим поиском по неопределенной последовательности. Каждый алгоритм, описанный в этом разделе, находится в заголовке `<algorithm>`.

lower_bound

Алгоритм `lower_bound` находит раздел в отсортированной последовательности.

Алгоритм возвращает итератор, соответствующий элементу `result`, который разделяет последовательность так, что элементы до результата меньше значения, тогда как `result` и все элементы после него не меньше `value`.

```
ForwardIterator lower_bound(fwd_begin, fwd_end, value, [comp]);
```

Аргументы

- Пара `ForwardIterators`, `fwd_begin` и `fwd_end`, представляющая целевую последовательность.
- `value` для разделения целевой последовательности.
- Необязательный оператор сравнения `comp`.

Сложность

Логарифмическая. Если предоставить случайный итератор — `O(log N)`, где `N = distance(fwd_begin, fwd_end);` в противном случае — `O(N)`

Дополнительные требования

Целевая последовательность должна быть отсортирована в соответствии с `operator<` или `comp`, если он представлен.

Пример

```
#include <algorithm>

TEST_CASE("lower_bound") {
```

```
  vector<int> numbers{ 2, 4, 5, 6, 6, 9 }; ❶
  const auto result = lower_bound(numbers.begin(), numbers.end(), 5); ❷
  REQUIRE(result == numbers.begin() + 2); ❸
}
```

Сначала создается `vector` объектов `int` ❶. Затем вызывается `lower_bound` с этим `vector` в качестве целевой последовательности и `value`, равным 5 ❷. Результатом является третий элемент, 5 ❸. Элементы 2 и 4 меньше 5, тогда как элементы 5, 6, 6 и 9 — нет.

upper_bound

Алгоритм `upper_bound` находит раздел в отсортированной последовательности.

Алгоритм возвращает итератор, соответствующий элементу `result`, который является первым элементом в целевой последовательности, превышающим значение.

```
ForwardIterator upper_bound(fwd_begin, fwd_end, value, [comp]);
```

Аргументы

- Пара `ForwardIterators`, `fwd_begin` и `fwd_end`, представляющая целевую последовательность.
- `value` для разделения целевой последовательности.
- Необязательный оператор сравнения `comp`.

Сложность

Логарифмическая. Если предоставить случайный итератор — `O(log N)`, где `N =distance (fwd_begin, fwd_end)`; в противном случае — `O(N)`

Дополнительные требования

Целевая последовательность должна быть отсортирована в соответствии с `operator<` или `comp`, если это предусмотрено.

Пример

```
#include <algorithm>

TEST_CASE("upper_bound") {
  vector<int> numbers{ 2, 4, 5, 6, 6, 9 }; ❶
  const auto result = upper_bound(numbers.begin(), numbers.end(), 5); ❷
  REQUIRE(result == numbers.begin() + 3); ❸
}
```

Сначала создается вектор объектов `int` ❶. Затем вызывается `upper_bound` с этим `vector` в качестве целевой последовательности и `value`, равным 5 ❷. Результатом является четвертый элемент 6, который является первым элементом в целевой последовательности, превышающим `value` ❸.

equal_range

Алгоритм `equal_range` находит диапазон определенных элементов в отсортированной последовательности.

Алгоритм возвращает `std::pair` итераторов, соответствующих полуоткрытому диапазону, равному `value`.

```
ForwardIteratorPair equal_range(fwd_begin, fwd_end, value, [comp]);
```

Аргументы

- Пара `ForwardIterators`, `fwd_begin` и `fwd_end`, представляющая целевую последовательность.
- `value` для поиска.
- Необязательный оператор сравнения `comp`.

Сложность

Логарифмическая. Если предоставить случайный итератор — `O(log N)`, где `N = distance(fwd_begin, fwd_end)`; в противном случае — `O(N)`.

Дополнительные требования

Целевая последовательность должна быть отсортирована в соответствии с `operator<` или `comp`, если он представлен.

Пример

```
#include <algorithm>

TEST_CASE("equal_range") {
  vector<int> numbers{ 2, 4, 5, 6, 6, 9 }; ❶
  const auto[rbeg, rend] = equal_range(numbers.begin(), numbers.end(), 6); ❷
  REQUIRE(rbeg == numbers.begin() + 3); ❸
  REQUIRE(rend == numbers.begin() + 5); ❹
}
```

Сначала создается вектор объектов `int` ❶. Затем вызывается `equal_range` с этим `vector` в качестве целевой последовательности и `value`, равным 6 ❷. В результате получается пара итераторов, представляющая соответствующий диапазон. Начальный итератор указывает на четвертый элемент ❸, а второй итератор указывает на шестой элемент ❹.

binary_search

Алгоритм `binary_search` находит определенный элемент в отсортированной последовательности.

Алгоритм возвращает `true`, если диапазон содержит значение. В частности он возвращает `true`, если целевая последовательность содержит такой элемент `x`, что ни `x < value`, ни `value <x`. Если `comp` предоставлен, он возвращает `true`, если целевая последовательность содержит такой элемент `x`, что не `comp(x, value)` или `comp(value, x)`.

```
bool binary_search(fwd_begin, fwd_end, value, [comp]);
```

Аргументы

- Пара `ForwardIterators`, `fwd_begin` и `fwd_end`, представляющая целевую последовательность.
- `value` для поиска.
- Необязательный оператор сравнения, `comp`.

Сложность

Логарифмическая. Если предоставить случайный итератор, `O(log N)`, где `N = distance(fwd_begin, fwd_end)`; в противном случае — `O(N)`.

Дополнительные требования

Целевая последовательность должна быть отсортирована в соответствии с `operator<` или `comp`, если это предусмотрено.

Пример

```
#include <algorithm>

TEST_CASE("binary_search") {
  vector<int> numbers{ 2, 4, 5, 6, 6, 9 }; ❶
  REQUIRE(binary_search(numbers.begin(), numbers.end(), 6)); ❷
  REQUIRE_FALSE(binary_search(numbers.begin(), numbers.end(), 7)); ❸
}
```

Сначала создается вектор объектов `int` ❶. Затем вызывается `binary_search` с этим `vector` в качестве целевой последовательности и значением 6. Поскольку последовательность содержит 6, `binary_search` возвращает `true` ❷. При вызове `binary_search` с 7 он возвращает `false`, поскольку целевая последовательность не содержит 7 ❸.

Алгоритмы разбиения

Секционированная последовательность содержит две смежные отдельные группы элементов. Эти группы не перекрываются, и первый элемент второй отдельной группы называется *точкой разделения*. stdlib содержит алгоритмы разбиения последовательностей, определения того, разбита ли последовательность на секции, и поиска точек разбиения. Каждый алгоритм, описанный в этом разделе, находится в заголовке `<algorithm>`.

is_partitioned

Алгоритм `is_partitioned` определяет, произошло ли разбиение последовательности.

ПРИМЕЧАНИЕ

Последовательность разбивается, если все элементы с некоторым атрибутом появляются перед элементами без атрибута.

Алгоритм возвращает `true`, если каждый элемент в целевой последовательности, для которого `pred` оценивается как `true`, появляется перед другими элементами.

```
bool is_partitioned([ep], ipt_begin, ipt_end, pred);
```

Аргументы

- Необязательная политика выполнения `std::execution`, `ep` (по умолчанию: `std::execution::seq`).
- Пара объектов `InputIterator`, `ipt_begin` и `ipt_end`, представляющих целевую последовательность.
- Предикат `pred`, определяющий членство в группе.

Сложность

Линейная. Максимум `distance(ipt_begin, ipt_end)` вычислений `pred`.

Примеры

```
#include <algorithm>

TEST_CASE("is_partitioned") {
  auto is_odd = [](auto x) { return x % 2 == 1; }; ❶

  vector<int> numbers1{ 9, 5, 9, 6, 4, 2 }; ❷
  REQUIRE(is_partitioned(numbers1.begin(), numbers1.end(), is_odd)); ❸

  vector<int> numbers2{ 9, 4, 9, 6, 4, 2 }; ❹
  REQUIRE_FALSE(is_partitioned(numbers2.begin(), numbers2.end(), is_odd)); ❺
}
```

Сначала создается лямбда-предикат с названием `is_odd`, который возвращает `true`, если заданное число нечетное ❶. Затем создается вектор объектов `int` ❷ и вызывается `is_partitioned` с этим `vector` в качестве целевой последовательности и `is_odd` в качестве предиката. Поскольку последовательность содержит все нечетные числа, помещенные перед ее четными номерами, `is_partitioned` возвращает `true` ❸.

Затем создается другой вектор объектов `int` ❹ и снова вызывается `is_partitioned` с этим вектором в качестве целевой последовательности и `is_odd` в качестве предиката. Поскольку последовательность не содержит всех своих нечетных чисел, помещенных перед ее четными числами (4 является четным и перед вторым 9), `is_partitioned` возвращает `false` ❺.

partition

Алгоритм `partition` разбивает последовательность.

Алгоритм изменяет целевую последовательность так, чтобы она была разделена в соответствии с `pred`. Возвращает точку разбиения. Первоначальный порядок элементов не обязательно сохраняется.

```
ForwardIterator partition([ep], fwd_begin, fwd_end, pred);
```

Аргументы

- Необязательная политика выполнения `std::execution`, `ep` (по умолчанию: `std::execution::seq`).
- Пара `ForwardIterators`, `fwd_begin` и `fwd_end`, представляющая целевую последовательность.
- Предикат `pred`, определяющий членство в группе.

Сложность

Линейная. Максимум `distance(fwd_begin, fwd_end)` вычислений `pred`.

Дополнительные требования

Элементы целевой последовательности должны быть заменяемыми.

Пример

```
#include <algorithm>

TEST_CASE("partition") {
  auto is_odd = [](auto x) { return x % 2 == 1; }; ❶
  vector<int> numbers{ 1, 2, 3, 4, 5 }; ❷
  const auto partition_point = partition(numbers.begin(),
                                         numbers.end(), is_odd); ❸
  REQUIRE(is_partitioned(numbers.begin(), numbers.end(), is_odd)); ❹
  REQUIRE(partition_point == numbers.begin() + 3); ❺
}
```

Сначала создается лямбда-предикат с именем `is_odd`, который возвращает `true`, если заданное число нечетное ❶. Затем создается вектор объектов `int` ❷ и вызывается `partition` с этим вектором в качестве целевой последовательности и `is_odd` в качестве предиката. Назначается результирующая точка разбиения в `partition_point` ❸.

При вызове `is_partitioned` в целевой последовательности с `is_odd` в качестве предиката он возвращает `true` ❹. В соответствии со спецификацией алгоритма *нельзя полагаться на порядок внутри групп*, но точка разделения всегда будет четвертым элементом, потому что целевая последовательность содержит три нечетных числа ❺.

partition_copy

Алгоритм `partition_copy` разделяет последовательность.

Алгоритм разделяет целевую последовательность, вычисляя `pred` для каждого элемента. Все истинные элементы копируются в `opt_true`, а все ложные элементы копируются в `opt_false`.

```
ForwardIteratorPair partition_copy([ep], ipt_begin, ipt_end,
                                   opt_true, opt_false, pred);
```

Аргументы

- Необязательная политика выполнения `std::execution`, `ep` (по умолчанию: `std::execution::seq`).
- Пара объектов `InputIterator`, `ipt_begin` и `ipt_end`, представляющих целевую последовательность.
- `OutputIterator`, `opt_true`, для получения копий истинных элементов.
- `OutputIterator`, `opt_false`, для получения копий ложных элементов.
- Предикат `pred`, определяющий членство в группе.

Сложность

Линейная. Ровно `distance(ipt_begin, ipt_end)` вычислений `pred`.

Дополнительные требования

- Элементы целевой последовательности должны иметь возможность присваивания копии.
- Диапазоны входа и выхода не должны перекрываться.

Пример

```
#include <algorithm>

TEST_CASE("partition_copy") {
  auto is_odd = [](auto x) { return x % 2 == 1; }; ❶
  vector<int> numbers{ 1, 2, 3, 4, 5 }, odds, evens; ❷
  partition_copy(numbers.begin(), numbers.end(),
                 back_inserter(odds), back_inserter(evens), is_odd); ❸
  REQUIRE(all_of(odds.begin(), odds.end(), is_odd)); ❹
  REQUIRE(none_of(evens.begin(), evens.end(), is_odd)); ❺
}
```

Сначала создается лямбда-предикат с именем `is_odd`, который возвращает `true`, если заданное число нечетное ❶. Затем создается вектор объектов `int`, содержащий числа от 1 до 5 и два пустых вектора, которые называются `odds` и `evens` ❷.

Затем вызывается `partition_copy` с числами в качестве целевой последовательности, `back_inserter` в `odds` в качестве вывода для истинных элементов, `back_inserter` в `evens` в качестве вывода для ложных элементов и `is_odd` в качестве предиката ❸. В результате все элементы в `odds` нечетные ❹, и ни один из элементов в `evens` не является нечетным ❺.

stable_partition

Алгоритм `stable_partition` стабильно разбивает последовательность.

ПРИМЕЧАНИЕ

Для стабильного разбиения может потребоваться больше вычислений, чем для нестабильного, поэтому пользователю предоставляется выбор.

Алгоритм изменяет целевую последовательность так, чтобы она была разделена в соответствии с `pred`. Возвращает точку разбиения. Оригинальный порядок элементов сохраняется.

```
BidirectionalIterator stable_partition([ep], bid_begin, bid_end, pred);
```

Аргументы

- Необязательная политика выполнения `std::execution`, `ep` (по умолчанию: `std::execution::seq`).
- Пара `BidirectionalIterators`, `bid_begin` и `bid_end`, представляющих целевую последовательность.
- Предикат `pred`, определяющий членство в группе.

Сложность

Квазилинейная. `O(N log N)`, где `N = distance(bid_begin, bid_end)`, или `O(N)`, если доступно достаточно памяти.

Дополнительные требования

Элементы целевой последовательности должны быть заменяемыми, перемещаемыми и присваиваемыми.

Пример

```
#include <algorithm>

TEST_CASE("stable_partition") {
  auto is_odd = [](auto x) { return x % 2 == 1; }; ❶
  vector<int> numbers{ 1, 2, 3, 4, 5 }; ❷
  stable_partition(numbers.begin(), numbers.end(), is_odd); ❸
  REQUIRE(numbers == vector<int>{ 1, 3, 5, 2, 4 }); ❹
}
```

Сначала создается лямбда-предикат `is_odd`, который возвращает `true`, если заданное число нечетное ❶. Затем создается вектор объектов `int` ❷ и вызывается `stable_partition` с этим `vector` в качестве целевой последовательности и `is_odd` в качестве предиката ❸. В результате вектор содержит элементы 1, 3, 5, 2, 4, так как это единственный способ разбить эти числа при сохранении их исходного порядка внутри группы ❹.

Алгоритмы слияния

Алгоритмы слияния объединяют две отсортированные целевые последовательности так, что результирующая последовательность содержит копии обеих целевых последовательностей и так же сортируется. Каждый алгоритм, описанный в этом разделе, находится в заголовке `<algorithm>`.

merge

Алгоритм `merge` объединяет две отсортированные последовательности.

Алгоритм копирует обе целевые последовательности в последовательность назначения. Последовательность назначения сортируется в соответствии с `operator<` или `comp`, если он представлен.

```
OutputIterator merge([ep], ipt_begin1, ipt_end1,
                     ipt_begin2, ipt_end2, opt_result, [comp]);
```

Аргументы

- Необязательная политика выполнения `std::execution`, `ep` (по умолчанию: `std::execution::seq`).
- Две пары `InputIterators`, `ipt_begin` и `ipt_end`, представляющие целевые последовательности.
- `OutputIterator`, `opt_result`, представляющий целевую последовательность.
- Предикат `pred`, который определяет членство в группе.

Сложность

Линейная. Максимум `N-1` сравнений, где `N = distance(ipt_begin1, ipt_end1) +distance(ipt_begin2, ipt_end2)`.

Дополнительные требования

Целевые последовательности должны быть отсортированы в соответствии с `operator<` или `comp`, если это предусмотрено.

Пример

```
#include <algorithm>

TEST_CASE("merge") {
  vector<int> numbers1{ 1, 4, 5 }, numbers2{ 2, 3, 3, 6 }, result; ❶
  merge(numbers1.begin(), numbers1.end(),
        numbers2.begin(), numbers2.end(),
        back_inserter(result)); ❷
  REQUIRE(result == vector<int>{ 1, 2, 3, 3, 4, 5, 6 }); ❸
}
```

Создается три `vector`-объекта: два содержащих отсортированные объекты типа `int`, а оставшийся — пустой ❶. Затем непустые векторы объединяются, а пустой вектор используется в качестве последовательности назначения с помощью `back_inserter` ❷. `result` содержит копии всех элементов из исходных последовательностей, и он также сортируется с помощью ❸.

Алгоритмы предельных значений

Несколько алгоритмов, называемых *алгоритмами предельных значений*, определяют минимальные и максимальные элементы или устанавливают ограничения на минимальное или максимальное значение элемента. Каждый алгоритм, описанный в этом разделе, находится в заголовке `<algorithm>`.

min и max

Алгоритм `min` или `max` определяет экстремумы последовательности.

Алгоритмы используют `operator<` или `comp` и возвращают минимальный (`min`) или максимальный (`max`) объект. Алгоритм `minmax` возвращает оба в виде `std::pair`, где `first` — минимум, а `second` — максимум.

```
T min(obj1, obj2, [comp]);
T min(init_list, [comp]);
T max(obj1, obj2, [comp]);
T max(init_list, [comp]);
Pair minmax(obj1, obj2, [comp]);
Pair minmax(init_list, [comp]);
```

Аргументы

- Два объекта, `obj1` и `obj2`, или
- Список инициализаторов, `init_list`, представляющий объекты для сравнения.
- Необязательная функция сравнения `comp`.

Сложность

Постоянная или линейная. Для перегрузок, принимающих `obj1` и `obj2`, ровно одно сравнение. Для списка инициализаторов не более `N-1` сравнений, где `N` — длина списка инициализаторов. В случае `minmax`, учитывая список инициализатора, это значение увеличивается до `3/2 N`.

Дополнительные требования

Элементы должны быть копируемыми и сравнимыми с использованием данного сравнения.

Примеры

```
#include <algorithm>

TEST_CASE("max and min") {
  auto length_compare = [](const auto& x1, const auto& x2) { ❶
    return x1.length() < x2.length();
  };

  string undisc="undiscriminativeness", vermin="vermin";
  REQUIRE(min(undisc, vermin, length_compare) == "vermin"); ❷

  string maxim="maxim", ultra="ultramaximal";
  REQUIRE(max(maxim, ultra, length_compare) == "ultramaximal"); ❸

  string mini="minimaxes", maxi="maximin";
  const auto result = minmax(mini, maxi, length_compare); ❹
  REQUIRE(result.first == maxi); ❺
  REQUIRE(result.second == mini); ❻
}
```

Сначала инициализируется лямбда-выражение с именем `length_compare`, которое использует `operator<` для сравнения длин двух вводов ❶. Затем используется `min`, чтобы определить, что из *undiscriminativeness* или *vermin* имеет меньшую длину ❷, и используется `max`, чтобы определить, что из *maxim* или *ultramaximal* имеет большую длину ❸. Наконец, используется `minmax`, чтобы определить, какой из *minimaxes* и *maximin* имеет минимальную и максимальную длину ❹. В результате получается пара ❺ ❻.

min_element и max_element

Алгоритмы `min_element` и `max_element` определяют экстремумы последовательности.

Алгоритмы используют `operator<` или `comp` и возвращают итератор, указывающий на минимальный (`min_element`) или максимальный (`max_element`) объект. Алгоритм `minimax_element` возвращает оба в виде `std::pair`, где `first` — минимум, а `second` — максимум.

```
ForwardIterator min_element([ep], fwd_begin, fwd_end, [comp]);
ForwardIterator max_element([ep], fwd_begin, fwd_end, [comp]);
Pair minmax_element([ep], fwd_begin, fwd_end, [comp]);
```

Аргументы

- Необязательная политика выполнения `std::execution`, `ep` (по умолчанию: `std::execution::seq`).
- Пара `ForwardIterators`, `fwd_begin` и `fwd_end`, представляющая целевую последовательность.
- Необязательная функция сравнения `comp`.

Сложность

Линейная. Для `max` и `min` не более `N-1` сравнений, где `N = distance(fwd _begin, fwd_end)`; для `minmax` ~`3/2 N`.

Дополнительные требования

Элементы должны быть сравнимы с использованием данной операции.

Примеры

```
#include <algorithm>

TEST_CASE("min and max element") {
  auto length_compare = [](const auto& x1, const auto& x2) { ❶
    return x1.length() < x2.length();
  };

  vector<string> words{ "civic", "deed", "kayak", "malayalam" }; ❷

  REQUIRE(*min_element(words.begin(), words.end(),
                       length_compare) == "deed"); ❸
  REQUIRE(*max_element(words.begin(), words.end(),
                       length_compare) == "malayalam"); ❹

  const auto result = minmax_element(words.begin(), words.end(),
                                     length_compare); ❺
  REQUIRE(*result.first == "deed"); ❻
  REQUIRE(*result.second == "malayalam"); ❼
}
```

Сначала инициализируется лямбда-предикат с именем `length_compare`, который использует `operator<` для сравнения длин двух вводов ❶. Затем инициализируется `vector` объектов `string` под названием `words`, содержащий четыре слова ❷. `min_element` используется для определения наименьшего из этих слов, передавая его в качестве целевой последовательности, а `length_compare` используется в качестве функции сравнения (`deed`) ❸, и `max_element` используется для определения самого большого элемента (`malayalam`) ❹. Наконец, используется `minmax_element`, кото-

рый возвращает оба как `std::pair` ❺. `first` относится к самому короткому `word` ❻, а `second` относится к самому длинному ❼.

clamp

Алгоритм `clamp` ограничивает значение.

Алгоритм использует `operator<` или `comp`, чтобы определить, находится ли `obj` в пределах от `low` до `high`. Если это так, алгоритм просто возвращает `obj`; в противном случае, если `obj` меньше, чем `low`, он возвращает `low`. Если `obj` больше, чем `high`, он возвращает `high`.

```
T& clamp(obj, low, high, [comp]);
```

Аргументы

- Объект `obj`.
- Объекты `low` и `high`.
- Дополнительная функция сравнения `comp`.

Сложность

Постоянная. Максимум два сравнения.

Дополнительные требования

Объекты должны быть сравнимы с использованием данной операции.

Примеры

```
#include <algorithm>

TEST_CASE("clamp") {
  REQUIRE(clamp(9000, 0, 100) == 100); ❶
  REQUIRE(clamp(-123, 0, 100) == 0); ❷
  REQUIRE(clamp(3.14, 0., 100.) == Approx(3.14)); ❸
}
```

В первом примере в функцию передается 9000 с интервалом от 0 до 100 включительно. Поскольку 9000> 100, результат равен `100` ❶. Во втором примере передается `-123` с тем же интервалом. Поскольку –123 <0, результат равен `0` ❷. Наконец, передается `3.14`, а поскольку он находится в пределах интервала, результат равен `3.14` ❸.

Числовые операции

Заголовок `<numeric>` обсуждался в главе 12, когда мы изучали его математические типы и функции. Он также предоставляет алгоритмы, хорошо подходящие для числовых операций. В этом разделе представлены многие из них. Каждый алгоритм, описанный в этом разделе, находится в заголовке `<numeric>`.

Полезные операторы

Некоторые числовые операции stdlib позволяют передавать оператор для настройки поведения. Для удобства заголовок `<functional>` предоставляет следующие шаблоны классов, которые предоставляют различные бинарные арифметические операции через `operator(T x, T y)`:

- `plus<T>` реализует сложение `x + y`;
- `minus<T>` реализует вычитание `x - y`;
- `multiplies<T>` реализует умножение `x * y`;
- `divides<T>` реализует деление `x/y`;
- `modulus<T>` реализует деление по модулю `x% y`.

Можно сложить два числа, используя шаблон `plus`.

Пример

```
#include <functional>

TEST_CASE("plus") {
  plus<short> adder; ❶
  REQUIRE(3 == adder(1, 2)); ❷
  REQUIRE(3 == plus<short>{}(1,2)); ❸
}
```

Сначала создается экземпляр `plus` с именем `adder` ❶, а затем он вызывается со значениями 1 и 2, что дает 3 ❷. Также можно полностью пропустить переменную и просто использовать вновь созданный `plus` напрямую для достижения того же результата ❸.

ПРИМЕЧАНИЕ

Обычно эти типы операторов не используются, если не используется общий код, который требует их наличия.

iota

Алгоритм `iota` заполняет последовательность инкрементными значениями.

Алгоритм присваивает инкрементные значения, начиная с начала, целевой последовательности.

```
void iota(fwd_begin, fwd_end, start);
```

Аргументы

- Пара итераторов, `fwd_begin` и `fwd_end`, представляющих целевую последовательность.
- Начальное значение.

Сложность

Линейная. `N` добавлений и присваиваний, где `N = distance(fwd_begin, fwd_end)`.

Дополнительные требования

Объекты должны быть присваиваемыми переменной `start`.

Пример

```
#include <numeric>
#include <array>

TEST_CASE("iota") {
  array<int, 3> easy_as; ❶
  iota(easy_as.begin(), easy_as.end(), 1); ❷
  REQUIRE(easy_as == array<int, 3>{ 1, 2, 3 }); ❸
}
```

Сначала инициализируется массив объектов `int` длиной 3 ❶. Затем вызывается `iota` с массивом в качестве целевой последовательности и 1 в качестве начального значения ❷. Результат — `array` с элементами 1, 2 и 3 ❸.

accumulate

Алгоритм `accumulate` сворачивает последовательность (по порядку).

ПРИМЕЧАНИЕ

Свернуть последовательность означает применить определенную операцию к элементам последовательности, передав общий результат следующей операции.

Алгоритм применяет `op` к `start` и первому элементу целевой последовательности. Он берет результат и снова применяет `op` к следующему элементу целевой последовательности, продолжая таким образом, пока не достигнет каждого элемента в целевой последовательности. В общем, этот алгоритм добавляет элементы целевой последовательности и начальное значение и возвращает результат.

```
T accumulate(ipt_begin, ipt_end, start, [op]);
```

Аргументы

- Пара итераторов, `ipt_begin` и `ipt_end`, представляющих целевую последовательность.
- Начальное значение.
- Необязательный бинарный оператор `op`, по умолчанию `plus`.

Сложность

Линейная. `N` выполнений операции, где `N=distance(ipt_begin, ipt_end)`.

Дополнительные требования

Элементы целевой последовательности должны быть копируемыми.

Примеры

```
#include <numeric>

TEST_CASE("accumulate") {
  vector<int> nums{ 1, 2, 3 }; ❶
  const auto result1 = accumulate(nums.begin(), nums.end(), -1); ❷
  REQUIRE(result1 == 5); ❸

  const auto result2 = accumulate(nums.begin(), nums.end(),
                                  2, multiplies<>()); ❹
  REQUIRE(result2 == 12); ❺
}
```

Сначала инициализируется вектор объектов `int` длиной 3 ❶. Затем вызывается `accumulate` с `vector` в качестве целевой последовательности и -1 в качестве начального значения ❷. Результат равен -1 + 1 + 2 + 3 = 5 ❸.

Во втором примере используется та же целевая последовательность, но с начальным значением 2 и оператором `multiplies` вместо этого ❹. Результат: 2 * 1 * 2 * 3 = 12 ❺.

reduce

Алгоритм `reduce` сворачивает последовательность (не обязательно по порядку).

Алгоритм идентичен `accumulate`, за исключением того что он принимает необязательный `execution` и не гарантирует порядок применений оператора.

```
T reduce([ep], ipt_begin, ipt_end, start, [op]);
```

Аргументы

- Необязательная политика выполнения `std::execution`, `ep` (по умолчанию: `std::execution::seq`).
- Пара итераторов `ipt_begin` и `ipt_end`, представляющих целевую последовательность.
- Начальное значение.
- Необязательный бинарный оператор, `op`, который по умолчанию `plus`.

Сложность

Линейная. `N` применений операции, где `N=distance(ipt_begin, ipt_end)`.

Дополнительные требования

- Элементы должны быть перемещаемыми, если `ep` опускается.
- Элементы должны быть копируемыми, если `ep` предоставлен.

Примеры

```
#include <numeric>

TEST_CASE("reduce") {
  vector<int> nums{ 1, 2, 3 }; ❶
  const auto result1 = reduce(nums.begin(), nums.end(), -1); ❷
  REQUIRE(result1 == 5); ❸

  const auto result2 = reduce(nums.begin(), nums.end(),
                              2, multiplies<>()); ❹
  REQUIRE(result2 == 12); ❺
}
```

Сначала инициализируется вектор объектов `int` длиной 3 ❶. Затем вызывается `reduce` с `vector` в качестве целевой последовательности и -1 в качестве начального значения ❷. Результат равен -1 + 1 + 2 + 3 = 5 ❸.

Во втором примере используется та же целевая последовательность, но с начальным значением 2 и оператором умножения вместо этого ❹. Результат: 2 * 1 * 2 * 3 = 12 ❺.

inner_product

Алгоритм `inner_product` вычисляет внутреннее произведение двух последовательностей.

ПРИМЕЧАНИЕ

Внутреннее произведение (или точечное произведение) — это скалярное значение, связанное с парой последовательностей.

Алгоритм применяет `op2` к каждой паре соответствующих элементов в целевой последовательности и суммирует их с `start`, используя `op1`.

```
T inner_product([ep], ipt_begin1, ipt_end1, ipt_begin2, start, [op1], [op2]);
```

Аргументы

- Пара итераторов `ipt_begin1` и `ipt_end1`, представляющих целевую последовательность 1.
- Итератор `ipt_begin2`, представляющий целевую последовательность 2.
- Начальное значение.
- Два необязательных бинарных оператора, `op1` и `op2`, которые по умолчанию равны `plus` и `multiply`.

Сложность

Линейная. N применений op1 и op2, где N=distance(ipt_begin1, ipt_end1).

Дополнительные требования

Элементы должны быть копируемыми.

Пример

```
#include <numeric>

TEST_CASE("inner_product") {
  vector<int> nums1{ 1, 2, 3, 4, 5 }; ❶
  vector<int> nums2{ 1, 0,-1, 0, 1 }; ❷
  const auto result = inner_product(nums1.begin(), nums1.end(),
                                    nums2.begin(), 10); ❸
  REQUIRE(result == 13); ❹
}
```

Сначала инициализируются два вектора с объектами int ❶ ❷. Затем вызывается inner_product с двумя векторными объектами в качестве целевых последовательностей и 10 в качестве начального значения ❸. Результат равен 10 + 1 * 1 + 2 * 0 + 3 * 1 + 4 * 0 + 4 * 1 = 13 ❹.

adjacent_difference

Алгоритм adjing_difference генерирует смежные различия.

ПРИМЕЧАНИЕ

Смежное различие является результатом применения некоторой операции к каждой паре соседних элементов.

Алгоритм устанавливает первый элемент последовательности назначения равным первому элементу целевой последовательности. Для каждого последующего элемента он применяет op к предыдущему элементу и текущему элементу и записывает возвращаемое значение в результат. Алгоритм возвращает конец последовательности назначения.

```
OutputIterator adjacent_difference([ep], ipt_begin, ipt_end, result, [op]);
```

Аргументы

- Пара итераторов, ipt_begin и ipt_end, представляющих целевую последовательность.
- Итератор result, представляющий последовательность назначения.
- Необязательный бинарный оператор op, по умолчанию minus.

Сложность

Линейная. `N-1` применений операции, где `N=distance(ipt_begin, ipt_end)`.

Дополнительные требования

- Элементы должны быть перемещаемыми, если `ep` опускается.
- Элементы должны быть копируемыми, если `ep` предоставлен.

Пример

```
#include <numeric>

TEST_CASE("adjacent_difference") {
  vector<int> fib{ 1, 1, 2, 3, 5, 8 }, fib_diff; ❶
  adjacent_difference(fib.begin(), fib.end(), back_inserter(fib_diff)); ❷
  REQUIRE(fib_diff == vector<int>{ 1, 0, 1, 1, 2, 3 }); ❸
}
```

Сначала инициализируются два вектора объектов `int`, один из которых содержит первые шесть чисел последовательности Фибоначчи, а другой — пустой ❶. Затем вызывается `adjacent_difference` с двумя векторами в качестве целевых последовательностей ❷. Результат является ожидаемым: первый элемент равен первому элементу последовательности Фибоначчи, а следующие элементы — смежные различия (1 – 1 = 0), (2 – 1 = 1), (3 – 2 = 1), (5 – 3 = 2), (8 – 5 = 3) ❸.

partial_sum

Алгоритм `partial_sum` генерирует частичные суммы.

Алгоритм устанавливает накопитель, равный первому элементу целевой последовательности. Для каждого последующего элемента целевой последовательности алгоритм добавляет этот элемент в накопитель и затем записывает накопитель в целевую последовательность. Алгоритм возвращает конец последовательности назначения.

```
OutputIterator partial_sum(ipt_begin, ipt_end, result, [op]);
```

Аргументы

- Пара итераторов, `ipt_begin` и `ipt_end`, представляющая целевую последовательность.
- Итератор `result`, представляющий последовательность назначения.
- Необязательный бинарный оператор `op`, по умолчанию `plus`.

Сложность

Линейная. `N-1` применений операции, где `N=distance(ipt_begin, ipt_end)`.

Пример

```
#include <numeric>

TEST_CASE("partial_sum") {
  vector<int> num{ 1, 2, 3, 4 }, result; ❶
  partial_sum(num.begin(), num.end(), back_inserter(result)); ❷
  REQUIRE(result == vector<int>{ 1, 3, 6, 10 }); ❸
}
```

Сначала инициализируются два вектора `int`, один из которых называется `num`, содержащий первые четыре счетчика, а пустой называется `result` ❶. Затем вызывается `partial_sum` с `num` в качестве целевой последовательности и `result` в качестве назначения ❷. Первый элемент равен первому элементу целевой последовательности, а следующие элементы являются частичными суммами (1 + 2 = 3), (3 + 3 = 6), (6 + 4 = 10) ❸.

Другие алгоритмы

Чтобы длинная глава не стала еще длиннее, многие алгоритмы опущены. Этот раздел предоставляет их обзор.

Операции с (максимальной) кучей

Диапазон длины `N` является максимальной кучей, если для всех `0 <i <N`, где $\frac{i-2}{2}$-й элемент (округленный в меньшую сторону) не меньше при сравнении, чем i-й элемент. Эти структуры обладают высокими характеристиками производительности в ситуациях, когда поиск и вставка максимального элемента должны быть быстрыми.

Заголовок `<algorithm>` содержит функции, которые полезны для обработки таких диапазонов, что показаны в таблице 18.1. См. [alg.heap.operations] для получения дополнительной информации.

Таблица 18.1. Алгоритмы, связанные с кучей в заголовке `<algorithm>`

Алгоритм	Описание
`is_heap`	Проверяет, является ли диапазон максимальной кучей
`is_heap_until`	Находит самый большой поддиапазон, который является максимальной кучей
`make_heap`	Создает максимальную кучу
`push_heap`	Добавляет элемент
`pop_heap`	Удаляет самый большой элемент
`sort_heap`	Превращает максимальную кучу в сортированный диапазон

Операции над множествами в сортированных диапазонах

Заголовок <algorithm> содержит функции, которые выполняют операции над множествами в сортированных диапазонах, например в таблице 18.2. См. [alg.set.operations] для получения дополнительной информации.

Таблица 18.2. Алгоритмы, связанные с множествами, в заголовке <algorithm>

Алгоритм	Описание
includes	Возвращает true, если один диапазон является подмножеством другого
set_difference	Вычисляет разницу между двумя множествами
set_intersection	Вычисляет пересечение двух множеств
set_symmetric_difference	Вычисляет симметричную разницу между двумя множествами
set_union	Вычисляет объединение двух множеств

Другие числовые алгоритмы

Заголовок <numeric> содержит еще несколько функций в дополнение к тем, которые представлены в разделе «Числовые операции». Таблица 18.3 перечисляет их. См. [numeric.ops] для получения дополнительной информации.

Таблица 18.3. Дополнительные числовые алгоритмы в заголовке <numeric>

Алгоритм	Описание
exclusive_scan	Работает как partial_sum, но исключает i-й элемент из i-й суммы
inclusive_scan	Работает как partial_sum, но выполняет перебор не по порядку и требует ассоциативной операции
transform_reduce	Применяет функциональный объект; затем удаляется из порядка
transform_exclusive_scan	Применяет функциональный объект; затем рассчитывает исключительное сканирование
transform_inclusive_scan	Применяет функциональный объект; затем рассчитывает включительное сканирование

Операции над памятью

Заголовок <memory> содержит ряд низкоуровневых функций для обработки неинициализированной памяти. Таблица 18.4 перечисляет их. См. [memory.syn] для получения дополнительной информации.

Boost Algorithm

Boost Algorithm — это большая библиотека алгоритмов, частично пересекающаяся со стандартной библиотекой. В целях экономии места в таблице 18.5 приведена только краткая ссылка на те алгоритмы, которых еще нет в стандартной библиотеке. Обратитесь к Boost Algorithm документации для получения дополнительной информации.

Таблица 18.4. Операции с неинициализированной памятью в заголовке `<memory>`

Алгоритм	Описание

`uninitialized_copy`
`uninitialized_copy_n`
`uninitialized_fill`
`uninitialized_fill_n`
	Копирует объекты в неинициализированную память

`uninitialized_move` `uninitialized_move_n`	Перемещает объекты в неинициализированную память

`uninitialized_default_construct`
`uninitialized_default_construct_n`
`uninitialized_value_construct`
`uninitialized_value_construct_n`
	Создает объекты в неинициализированной памяти

`destroy_at`
`destroy`
`destroy_n`
	Уничтожает объекты

Таблица 18.5. Дополнительные алгоритмы, доступные в Boost Algorithm

Алгоритм	Описание

`boyer_moore`
`boyer_moore_horspool`
`knuth_morris_pratt`
	Быстрые алгоритмы поиска последовательностей значений

`hex` `unhex`	Запись/чтение шестнадцатеричных символов
`gather`	Принимает последовательность и перемещает элементы, удовлетворяющие предикату, в заданную позицию
`find_not`	Находит первый элемент в последовательности, не равной значению
`find_backward`	Работает как `find`, но в обратном порядке

Продолжение

Таблица 18.5 *(продолжение)*

Алгоритм	Описание
`is_partitioned_until`	Возвращает конечный итератор для наибольшей секционированной подпоследовательности, которая начинается с первого элемента целевой последовательности
`apply_permutation` `apply_reverse_permutation`	Принимает последовательность элементов и порядок последовательности и переставляет последовательность элементов в соответствии с порядком
`is_palindrome`	Возвращает `true`, если последовательность является палиндромом

ЗАМЕЧАНИЕ О ДИАПАЗОНАХ

В главе 8 выражения диапазона представлены как часть цикла `for` на основе диапазона. Вспомните, что диапазон предоставляет методы `begin` и `end`, возвращающие итераторы. Поскольку можно устанавливать требования к итераторам для поддержки определенных операций, то также можно и устанавливать переходные требования к диапазонам, чтобы они предоставляли определенные итераторы. Каждый алгоритм имеет определенные эксплуатационные требования, которые отражаются в видах запрашиваемых итераторов. Поскольку можно инкапсулировать требования входной последовательности алгоритма в терминах диапазонов, важно знать различные типы диапазонов, чтобы понимать ограничения каждого алгоритма.

Диапазоны еще не являются официальной частью C++. Хотя вы все равно получите множество преимуществ от понимания взаимосвязи между диапазонами, итераторами и алгоритмами, есть два недостатка. Во-первых, для алгоритмов все еще требуются итераторы в качестве входных аргументов, поэтому, даже если диапазон находится под рукой, придется извлекать итераторы вручную (например, с помощью `begin` и `end`). Во-вторых, как и в случае с другими шаблонами функций, иногда получаются поразительно плохие сообщения об ошибках при нарушении требований алгоритма.

Ведется работа по формальному введению диапазонов в язык. Фактически концепты и диапазоны, скорее всего, войдут в стандарт C++ одновременно, потому что они так хорошо соответствуют друг другу.

Если вы хотите поэкспериментировать с одной возможной реализацией диапазонов, обратитесь к Boost Range.

Что еще почитать?

- Международный стандарт ИСО/МЭК (2017) — Язык программирования C++ (Международная организация по стандартизации; Женева, Швейцария; isocpp.org/std/the-standard/)
- «Стандартная библиотека C++. Справочное руководство», 2-е издание, Николай М. Джосаттис (Вильямс, 2017)
- «Algorithmic Complexity», Victor Adamchik (www.cs.cmu.edu/ ~adamchik/15-121/lectures/Algorithmic%20Complexity/complexity.html)
- «The Boost C++ Libraries», 2nd Edition, Boris Schäling (XML Press, 2014)

19 Конкурентность и параллелизм

У Старшей из Сторожевых псов была собственная излюбленная следи-фраза: «Покажите мне пример совершенно гладко идущей рутины, и я найду вам того, кто прикрывает ошибки. Реальный корабль, бывает, качает».

Фрэнк Херберт, «Дом глав родов Дюны»

В программировании *конкурентность* означает две или более задач, выполняемых в данный период времени. *Параллелизм* означает две или более задач, выполняющихся одновременно. Часто эти термины используются взаимозаменяемо без негативных последствий, потому что они так тесно связаны. Эта глава знакомит с основами обеих концепций. Поскольку конкурентное и параллельное программирование — это объемные и сложные темы, их тщательная проработка требует целой книги. Список таких книг вы найдете в разделе «Что еще почитать?» в конце этой главы.

В этой главе вы узнаете о параллельном и конкурентном программировании с фьючерсами. Изучите, как безопасно делиться данными, используя мьютексы, условные переменные и атомарность. Также в этой главе рассказано, как политики выполнения помогают ускорить код и меж тем подвергают его опасности.

Конкурентное программирование

Конкурентные программы имеют несколько *потоков выполнения* (или просто *потоков*), которые представляют собой последовательности инструкций. В большинстве

сред времени выполнения операционная система действует как планировщик, чтобы определить, когда поток выполняет свою следующую инструкцию. Каждый процесс может иметь один или несколько потоков, которые обычно совместно используют ресурсы, например память. Поскольку планировщик определяет, когда выполняются потоки, программист обычно не может полагаться на их порядок. В обмен программы могут выполнять несколько задач в один и тот же период времени (или в одно и то же время), что часто приводит к серьезному ускорению. Чтобы наблюдать любое ускорение от последовательной до конкурентной версии, системе потребуется конкурентное аппаратное обеспечение, например многоядерный процессор.

Этот раздел начинается с асинхронных задач — высокоуровневого метода обеспечения одновременной работы программ. Вы изучите некоторые основные методы координации между этими задачами, когда они обрабатывают общее изменяемое состояние.

Затем мы поговорим о некоторых средствах низкого уровня, доступных в stdlib, используемых в уникальных ситуациях, когда инструменты более высокого уровня не имеют требуемых характеристик производительности.

Асинхронные задачи

Одним из способов обеспечения конкурентности в программе является создание *асинхронных задач*. Асинхронная задача не требует немедленного результата. Чтобы запустить асинхронную задачу, используется шаблон функции `std::async` в заголовке `<future>`.

async

При вызове `std::async` первым аргументом является политика запуска `std::launch`, которая принимает одно из двух значений: `std::launch::async` или `std::launch::deferred`. Если передать `launch::async`, среда выполнения создает новый поток для запуска задачи. Если передать параметр `deferred`, среда выполнения дождется, пока не понадобится результат задачи, прежде чем выполнять ее (шаблон, иногда называемый отложенным вычислением). Этот первый аргумент является необязательным и по умолчанию используется `async|deferred`, что означает, что реализация зависит от используемой стратегии. Второй аргумент `std::async` — это функциональный объект, представляющий задачу, которую нужно выполнить. Нет никаких ограничений на количество или тип аргументов, которые принимает объект функции, и он может возвращать любой тип. Функция `std::async` является вариативным шаблоном с пакетом параметров функции. Любые дополнительные аргументы, которые передаются за пределы объекта функции, будут использоваться для вызова объекта функции при запуске асинхронной задачи. Кроме того, `std::async` возвращает объект, называемый `std::future`.

Следующее упрощенное объявление `async` обобщает вышесказанное:

```
std::future<FuncReturnType> std::async([policy], func, Args&&... args);
```

Теперь, когда вы знаете, как вызывать `async`, давайте посмотрим, как взаимодействовать с его возвращаемым значением.

Назад в будущее

`future` — это шаблон класса, который содержит значение асинхронной задачи. Он имеет единственный параметр шаблона, который соответствует типу возвращаемого значения асинхронной задачи. Например, если передать функциональный объект, который возвращает строку, `async` вернет `future<string>`. В будущем можно взаимодействовать с асинхронной задачей тремя способами.

Во-первых, можно запросить достоверность `future`, используя метод `valid`. С действительным `future` связано общее состояние. Асинхронные задачи имеют общее состояние, поэтому они могут передавать результаты. Любой `future`, возвращаемый `async`, будет действительным до тех пор, пока вы не получите возвращаемое значение асинхронной задачи, после чего срок жизни общего состояния заканчивается, как показано в листинге 19.1.

Листинг 19.1. Асинхронная функция возвращает правильное будущее

```
#include <future>
#include <string>

using namespace std;

TEST_CASE("async returns valid future") {
  using namespace literals::string_literals;
  auto the_future = async([] { return "female"s; }); ❶
  REQUIRE(the_future.valid()); ❷
}
```

В примере запускается асинхронная задача, которая просто возвращает `string` ❶. Поскольку `async` всегда возвращает действительный `future`, `valid` возвращает `true` ❷.

Если создавать `future` по умолчанию, оно не будет связано с общим состоянием, поэтому `valid` вернет `false`, как показано в листинге 19.2.

Листинг 19.2. Построенный по умолчанию future недопустим

```
TEST_CASE("future invalid by default") {
  future<bool> default_future; ❶
  REQUIRE_FALSE(default_future.valid()); ❷
}
```

`future` был создан по умолчанию ❶, и метод `valid` вернул `false` ❷.

Во-вторых, можно получить значение из допустимого `future` с помощью метода `get`. Если асинхронная задача еще не выполнена, вызов `get` блокирует текущий выполняемый поток, пока не станет доступен результат. В листинге 19.3 показано, как использовать `get` для получения возвращаемых значений.

Листинг 19.3. Функция async возвращает корректный future

```
TEST_CASE("async returns the return value of the function object") {
  using namespace literals::string_literals;
  auto the_future = async([] { return "female"s; }); ❶
  REQUIRE(the_future.get() == "female"); ❷
}
```

`async` используется для запуска асинхронной задачи ❶, а затем вызывается метод `get` для результирующего `future`. Как и ожидалось, результатом будет возвращаемое значение функционального объекта, который был передан `async` ❷.

Если асинхронная задача выдает исключение, `future` соберет это исключение и сгенерирует его при вызове `get`, как показано в листинге 19.4.

Листинг 19.4. Метод get генерирует исключение, выданное асинхронной задачей

```
TEST_CASE("get may throw ") {
  auto ghostrider = async(
                      [] { throw runtime_error{ "The pattern is full." }; }); ❶
  REQUIRE_THROWS_AS(ghostrider.get(), runtime_error); ❷
}
```

В `async` передается лямбда-выражение, которое выбрасывает `runtime_error` ❶. При вызове `get` выдается исключение ❷.

В-третьих, можно проверить, завершена ли асинхронная задача, используя либо `std::wait_for`, либо `std::wait_until`. Выбор зависит от вида объекта `chrono`, который нужно передать. Если существует объект `duration`, будет использоваться `wait_for`. Если существует объект `time_point`, будет использоваться `wait_until`. Оба возвращают `std::future_status`, который принимает одно из трех значений:

- `future_status::deferred` сигнализирует о том, что асинхронная задача будет выполняться лениво, поэтому задача будет выполнена после вызова `get`;
- `future_status::ready` указывает, что задача выполнена и результат готов;
- `future_status::timeout` указывает, что задача не готова.

Если задача завершится раньше указанного периода ожидания, `async` вернется раньше.

В листинге 19.5 показано, как использовать `wait_for` для проверки состояния асинхронной задачи.

Листинг 19.5. Проверка состояния асинхронной задачи с использованием wait_for

```
TEST_CASE("wait_for indicates whether a task is ready") {
  using namespace literals::chrono_literals;
  auto sleepy = async(launch::async, [] { this_thread::sleep_for(100ms); }); ❶
  const auto not_ready_yet = sleepy.wait_for(25ms); ❷
  REQUIRE(not_ready_yet == future_status::timeout); ❸
  const auto totally_ready = sleepy.wait_for(100ms); ❹
  REQUIRE(totally_ready == future_status::ready); ❺
}
```

Сначала запускается асинхронная задача с помощью `async`, который просто ждет до 100 миллисекунд, прежде чем вернуть значение ❶. Затем вызывается `wait_for` с 25 миллисекундами ❷. Поскольку задача все еще находится в спящем режиме (25 <100), `wait_for` возвращает `future_status::timeout` ❸. Снова вызывается `wait_for` и ожидает еще 100 миллисекунд ❹. Поскольку второй `wait_for` завершится после завершения асинхронной задачи, последний `wait_for` вернет `future_status::ready` ❺.

ПРИМЕЧАНИЕ

Технически не гарантируется, что утверждения из листинга 19.5 пройдут тест. В разделе «Ожидания» на с. 398 было введено `this_thread::sleep_for`, который не является точным. Операционная среда отвечает за планирование потоков и может планировать спящий поток позже указанной длительности.

Пример с асинхронными задачами

В листинге 19.6 содержится функция `factorize`, которая находит все целочисленные факторы.

ПРИМЕЧАНИЕ

Алгоритм факторизации в листинге 19.6 крайне неэффективен, но достаточно хорош для этого примера. За эффективными алгоритмами целочисленной факторизации обращайтесь к алгоритму Диксона, алгоритму непрерывной дробной дроби или к квадратичному решету.

Листинг 19.6. Очень простой алгоритм целочисленной факторизации

```
#include <set>

template <typename T>
std::multiset<T> factorize(T x) {
  std::set<T> result{ 1 }; ❶
  for(T candidate{ 2 }; candidate <= x; candidate++) { ❷
    if (x % candidate == 0) { ❸
      result.insert(candidate); ❹
      x /= candidate; ❺
      candidate = 1; ❻
    }
  }
  return result;
}
```

Алгоритм принимает один аргумент `x` и начинается с инициализации набора, содержащего 1 ❶. Затем он выполняет итерацию от 2 до `x` ❷, проверяя, приводит ли деление по модулю с `candidate` к 0 ❸. Если это так, кандидат является фактором, и он добавляется в набор факторов ❹. `x` делится на только что обнаруженный коэффициент ❺ и затем возобновляется поиск, сбрасывая `candidate` в 1 ❻.

Поскольку целочисленная факторизация является сложной проблемой (а листинг 19.6 настолько неэффективен), вызовы `factorize` могут занимать много времени по сравнению с большинством функций, с которыми вы столкнулись в этой книге.

Это делает его главным кандидатом для асинхронных задач. Функция `factor_task` в листинге 19.7 использует верный `Stopwatch` из листинга 12.25 в главе 12 для обертки `factorize` и возвращает красиво отформатированное сообщение.

Листинг 19.7. Функция factor_task, которая оборачивает вызов factorize и возвращает красиво отформатированное сообщение

```
#include <set>
#include <chrono>
#include <sstream>
#include <string>

using namespace std;

struct Stopwatch {
--пропуск--
};

template <typename T>
set<T> factorize(T x) {
--пропуск--
}

string factor_task(unsigned long x) { ❶
  chrono::nanoseconds elapsed_ns;
  set<unsigned long long> factors;
  {
    Stopwatch stopwatch{ elapsed_ns }; ❷
    factors = factorize(x); ❸
  }
  const auto elapsed_ms =
            chrono::duration_cast<chrono::milliseconds>(elapsed_ns).count(); ❹
  stringstream ss;
  ss << elapsed_ms << " ms: Factoring " << x << " ( "; ❺
  for(auto factor : factors) ss << factor << " "; ❻
  ss << ")\n";
  return ss.str(); ❼
}
```

Как и `factorize`, `factor_task` принимает один аргумент `x` для факторизации ❶. (Для простоты `factor_task` принимает `unsigned long`, а не шаблонный аргумент). Затем инициализируется `Stopwatch` во вложенной области действия ❷, а затем вызывается `factorize` с `x` ❸. В результате `elapsed_ns` содержит количество наносекунд, прошедших при выполнении факторизации, а коэффициент содержит все факторы `x`.

Затем создается красиво отформатированная строка, сначала преобразовав `elapsed_ns` в число в миллисекундах ❹. Эта информация записывается в объект потока строк с именем `ss` ❺, за которым следует множитель `x` ❻. Затем полученная строка возвращается ❼.

В листинге 19.8 используется `factor_task` для факторизации шести различных чисел и записи общего истекшего времени программы.

Листинг 19.8. Программа, использующая factor_task для факторизации шести различных чисел

```
#include <set>
#include <array>
#include <vector>
#include <iostream>
#include <limits>
#include <chrono>
#include <sstream>
#include <string>

using namespace std;

struct Stopwatch {
--пропуск--
};

template <typename T>
set<T> factorize(T x) {
--пропуск--
}

string factor_task(unsigned long long x) {
--пропуск--
}

array<unsigned long long, 6> numbers{ ❶
        9'699'690,
        179'426'549,
        1'000'000'007,
        4'294'967'291,
        4'294'967'296,
        1'307'674'368'000
};

int main() {
  chrono::nanoseconds elapsed_ns;
  {
    Stopwatch stopwatch{ elapsed_ns }; ❷
    for(auto number : numbers) ❸
      cout << factor_task(number); ❹
  }
  const auto elapsed_ms =
            chrono::duration_cast<chrono::milliseconds>(elapsed_ns).count(); ❺
  cout << elapsed_ms << "ms: total program time\n"; ❻
}
--------------------------------------------------------------------------------
0 ms: Factoring 9699690 ( 1 2 3 5 7 11 13 17 19 )
1274 ms: Factoring 179426549 ( 1 179426549 )
6804 ms: Factoring 1000000007 ( 1 1000000007 )
29035 ms: Factoring 4294967291 ( 1 4294967291 )
0 ms: Factoring 4294967296 ( 1 2 )
0 ms: Factoring 1307674368000 ( 1 2 3 5 7 11 13 )
37115ms: total program time
```

Создается массив, содержащий шесть чисел различного размера и простоты ❶. Затем инициализируется `Stopwatch` ❷, перебирается каждый элемент в `numbers` ❸ и вызывается `factor_task` с ними ❹. Затем определяется время выполнения программы в миллисекундах ❺ и выводится результат ❻.

Выходные данные показывают, что некоторые числа, такие как 9 699 690, 4 294 967 296 и 1 307 674 368 000, учитываются почти сразу, поскольку содержат небольшие множители. Однако простые числа занимают довольно много времени. Обратите внимание, что поскольку программа является однопоточной, время выполнения для всей программы примерно равно сумме времени, необходимого для факторизации каждого числа.

Что делать, если вы рассматриваете каждую `factor_task` как асинхронную задачу? В листинге 19.9 показано, как это сделать с помощью `async`.

Листинг 19.9. Программа, использующая factor_task для асинхронного разложения шести разных чисел

```
#include <set>
#include <vector>
#include <array>
#include <iostream>
#include <limits>
#include <chrono>
#include <future>
#include <sstream>
#include <string>

using namespace std;

struct Stopwatch {
--пропуск--
};

template <typename T>
set<T> factorize(T x) {
--пропуск--
}

string factor_task(unsigned long long x) {
--пропуск--
}

array<unsigned long long, 6> numbers{
--пропуск--
};

int main() {
  chrono::nanoseconds elapsed_ns;
  {
    Stopwatch stopwatch{ elapsed_ns }; ❶
    vector<future<string>> factor_tasks; ❷
```

```
    for(auto number : numbers) ❸
      factor_tasks.emplace_back(async(launch::async, factor_task, number)); ❹
    for(auto& task : factor_tasks) ❺
      cout << task.get(); ❻
  }
  const auto elapsed_ms =
            chrono::duration_cast<chrono::milliseconds>(elapsed_ns).count(); ❼
  cout << elapsed_ms << " ms: total program time\n"; ❽
}
--------------------------------------------------------------------------------
0 ms: Factoring 9699690 ( 1 2 3 5 7 11 13 17 19 )
1252 ms: Factoring 179426549 ( 1 179426549 )
6816 ms: Factoring 1000000007 ( 1 1000000007 )
28988 ms: Factoring 4294967291 ( 1 4294967291 )
0 ms: Factoring 4294967296 ( 1 2 )
0 ms: Factoring 1307674368000 ( 1 2 3 5 7 11 13 )
28989 ms: total program time
```

Как и в листинге 19.8, инициализируется `Stopwatch`, чтобы отслеживать продолжительность выполнения программы ❶. Затем инициализируется `vector` под названием `factor_tasks`, который содержит объекты типа `future<string>` ❷. Вы перебираете `numbers` ❸, вызывая `async` со стратегией `launch::async`, определяя `factor_task` в качестве функционального объекта и передавая `number` в качестве аргумента задачи. Вызывается `emplace_back` для каждого получающегося `future` в `factor_tasks` ❹. Теперь, когда `async` запустил каждую задачу, перебирается каждый элемент `factor_tasks` ❺, вызывается `get` для каждой задачи и записывается в `cout` ❻. После получения значения от всех фьючерсов определяется количество миллисекунд, необходимое для выполнения всех задач ❼, и записывается в `cout` ❽.

Благодаря многозадачности общее время программы в листинге 19.9 примерно равно максимальному времени выполнения задачи (28 988 мс), а не сумме времени выполнения задачи, как в листинге 19.8 (37 115 мс).

ПРИМЕЧАНИЕ

Время из листинга 19.8 и листинга 19.9 будет варьироваться от запуска к запуску.

Совместное использование и координирование

Конкурентное программирование с асинхронными задачами является простым, если задачи не требуют синхронизации и совместного использования изменяемых данных. Например, рассмотрим простую ситуацию, когда два потока обращаются к одному и тому же целому числу. Один поток будет увеличивать целое число, а другой — уменьшать его. Чтобы изменить переменную, каждый поток должен прочитать текущее значение переменной, выполнить операцию сложения или вычитания, а затем записать переменную в память. Без синхронизации два потока будут выполнять эти операции в неопределенном порядке чередования. Такие ситуации иногда называют *состоянием гонки*, потому что результат зависит от того, какой по-

ток выполняется первым. Листинг 19.10 показывает, насколько катастрофической является эта ситуация.

Листинг 19.10. Посмотрите, насколько разрушительным может быть несинхронизированный и изменчивый общий доступ к данным

```
#include <future>
#include <iostream>

using namespace std;

void goat_rodeo() {
  const size_t iterations{ 1'000'000 };
  int tin_cans_available{}; ❶
  auto eat_cans = async(launch::async, [&] { ❷
    for(size_t i{}; i<iterations; i++)
      tin_cans_available--; ❸
  });
  auto deposit_cans = async(launch::async, [&] { ❹
    for(size_t i{}; i<iterations; i++)
    tin_cans_available++; ❺
  });
  eat_cans.get(); ❻
  deposit_cans.get(); ❼
  cout << "Tin cans: " << tin_cans_available << "\n"; ❽
}

int main() {
  goat_rodeo();
  goat_rodeo();
  goat_rodeo();
}
--------------------------------------------------------------------------------
Tin cans: -609780
Tin cans: 185380
Tin cans: 993137
```

ПРИМЕЧАНИЕ

При каждом запуске программы в листинге 19.10 вы будете получать разные результаты, поскольку программа имеет неопределенное поведение.

Листинг 19.10 включает в себя определение функции `goat_rodeo`, которая описывает катастрофическое состояние гонки, и `main`, который вызывает `goat_rodeo` три раза. В `goat_rodeo` инициализируются общие данные `tin_cans_available` ❶. Затем запускается асинхронная задача под названием `eat_cans` ❷, в которой путешествие коз (goat) уменьшает общую переменную `tin_cans_available` в миллион раз ❸. Затем запускается еще одна асинхронная задача с именем `deposit_cans` ❹, в которой увеличивается `tin_cans_available` ❺. После запуска двух задач вы ожидаете их завершения, вызывая `get` (порядок не имеет значения) ❻ ❼. После выполнения задач выводится переменная `tin_cans_available` ❽.

Интуитивно можно ожидать, что `tin_cans_available` будет равняться нулю после завершения каждой задачи. В конце концов, независимо от того, как вызываются уменьшения и увеличения, если выполнить их в равном количестве, они будут отменены. `goat_rodeo` вызывается три раза, и каждый вызов дает совершенно другой результат.

В таблице 19.1 приведена одна из возможных причин, по которым несинхронизированный доступ в листинге 19.10 работает неправильно.

Таблица 19.1. Один из возможных запусков для `eat_cans` и `deposit_cans`

`eat_cans`	`deposit_cans`	`cans_available`
Читает `cans_available` (0)		`0`
	Читает `cans_available` (0) ❶	`0`
Вычисляет `cans_available+1` (1)		`0`
	Вычисляет `cans_available-1` (–1) ❸	`0`
Записывает `cans_available+1` (1) ❷		`1`
	Записывает `cans_available-1` (–1) ❹	`-1`

Таблица 19.1 показывает, как чередование операций чтения и записи приводит к катастрофе. В этом конкретном воплощении чтение по `deposit_cans` ❶ предшествует записи из `eat_cans` ❷, поэтому `deposit_cans` вычисляет устаревший результат ❸. Если он не так уж и плох, при записи происходит затирание записи из `eat_cans` ❹.

Основная проблема этой гонки данных — *несинхронизированный доступ к изменяемым общим данным*. Вы можете задаться вопросом, почему `cans_available` не обновляется немедленно, когда поток вычисляет `cans_available+1` или `cans_available-1`.

Ответ в том, что каждая из строк в таблице 19.1 представляет момент времени, когда какая-либо инструкция завершает выполнение, а инструкции для сложения, вычитания, чтения и записи памяти являются отдельными. Поскольку переменная `cans_available` является общей и оба потока пишут в нее без синхронизации своих действий, инструкции выполняются чередованием неопределенным образом во время выполнения (с катастрофическими результатами). В следующих подразделах вы изучите три инструмента для работы с такими ситуациями: мьютексы, переменные условия и атомарность.

Мьютексы

Алгоритм взаимного исключения (*мьютекс*) — это механизм, предотвращающий одновременный доступ нескольких потоков к ресурсам. Мьютексы — это *примитивы синхронизации*, которые поддерживают две операции: блокировать и разблокировать. Когда потоку необходим доступ к общим данным, он блокирует мьютекс. Эта

операция может предотвращаться в зависимости от природы мьютекса и наличия блокировки у другого потока. Когда потоку больше не нужен доступ, он разблокирует мьютекс.

Заголовок `<mutex>` предоставляет несколько опций мьютекса:

- `std::mutex` обеспечивает базовое взаимное исключение;
- `td::timed_mutex` обеспечивает взаимное исключение с тайм-аутом;
- `std::recursive_mutex` обеспечивает взаимное исключение, которое позволяет рекурсивную блокировку одним и тем же потоком;
- `std::recursive_timed_mutex` обеспечивает взаимное исключение, которое позволяет рекурсивную блокировку одним и тем же потоком и тайм-аут.

Заголовок `<shared_mutex>` предоставляет две дополнительные опции:

- `std::shared_mutex` предоставляет возможность совместного взаимного исключения, что означает, что несколько потоков могут одновременно владеть мьютексом. Эта опция обычно используется в сценариях, когда несколько читающих потоков могут получить доступ к общим данным, но записывающему потоку нужен эксклюзивный доступ;
- `std::shared_timed_mutex` обеспечивает общее средство взаимного исключения и реализует блокировку с тайм-аутом.

ПРИМЕЧАНИЕ

Для простоты эта глава охватывает только мьютекс. См. [thread.mutex] для получения дополнительной информации о других опциях.

Класс `mutex` определяет только один конструктор по умолчанию. Когда нужно получить взаимное исключение, вызывается один из двух методов объекта мьютекса: `lock` или `try_lock`. Если вызван `lock`, который не принимает аргументов и возвращает `void`, вызывающий поток блокируется, пока `mutex` не станет доступным. Если вызывается `try_lock`, который не принимает аргументов и возвращает `bool`, он немедленно возвращается. Если `try_lock` успешно получил взаимное исключение, он возвращает `true`, и вызывающий поток теперь владеет блокировкой. Если `try_lock` был неудачным, он возвращает `false`, и вызывающий поток не владеет блокировкой. Чтобы снять блокировку взаимного исключения, просто вызовите метод `unlock`, который не принимает аргументов и возвращает `void`.

Листинг 19.11 показывает основанный на блокировке способ разрешить состояние гонки в листинге 19.10.

Класс `mutex`, добавленный в `goat_rodeo` ❶ с именем `tin_can_mutex`, обеспечивает взаимное исключение для `tin_cans_available`. Внутри каждой асинхронной задачи поток получает блокировку ❷ ❹ перед изменением `tin_cans_available`. Как только поток завершит изменение, он разблокирует ❸ ❺. Обратите внимание, что итоговое количество `tin_cans_available`в конце каждого пробега равно нулю ❻ ❼ ❽, что свидетельствует о том, что состояние гонки исправлено.

Листинг 19.11. Использование мьютекса для разрешения состояния гонки в листинге 19.10

```
#include <future>
#include <iostream>
#include <mutex>

using namespace std;

void goat_rodeo() {
  const size_t iterations{ 1'000'000 };
  int tin_cans_available{};
  mutex tin_can_mutex; ❶
  auto eat_cans = async(launch::async, [&] {
    for(size_t i{}; i<iterations; i++) {
      tin_can_mutex.lock(); ❷
      tin_cans_available--;
      tin_can_mutex.unlock(); ❸
    }
  });
  auto deposit_cans = async(launch::async, [&] {
    for(size_t i{}; i<iterations; i++) {
      tin_can_mutex.lock(); ❹
      tin_cans_available++;
      tin_can_mutex.unlock(); ❺
    }
  });
  eat_cans.get();
  deposit_cans.get();
  cout << "Tin cans: " << tin_cans_available << "\n";
}

int main() {
  goat_rodeo(); ❻
  goat_rodeo(); ❼
  goat_rodeo(); ❽
}
--------------------------------------------------------------------------------
Tin cans: 0 ❻
Tin cans: 0 ❼
Tin cans: 0 ❽
```

Если вы думаете, что блокировка `mutex` — идеальная работа для RAII-объекта, то правы. Предположим, вы забыли вызвать `unlock` для мьютекса, например, потому что он вызвал исключение. Когда следующий поток придет и попытается получить мьютекс с помощью `lock`, программа остановится. По этой причине stdlib предоставляет классы RAII для обработки мьютексов в заголовке `<mutex>`. Там вы найдете несколько шаблонов классов, каждый из которых принимает мьютексы в качестве параметров конструктора, и параметр шаблона, соответствующий классу мьютексов:

- `std::lock_guard` — это некопируемая, непереносимая оболочка RAII, которая принимает объект мьютекса в своем конструкторе, где он вызывает блокировку. Затем вызывается `unlock` в деструкторе;

- `std::scoped_lock` — это тупик, избегающий оболочки RAII для нескольких мьютексов;
- `std::unique_lock` реализует оболочку владения перемещаемым мьютексом;
- `std::shared_lock` реализует переносимую оболочку владения разделяемым мьютексом.

> **РЕАЛИЗАЦИЯ МЬЮТЕКСОВ**
>
> На практике мьютексы реализуются несколькими способами. Возможно, самый простой мьютекс — это спин-блокировка, в которой поток будет выполнять цикл до тех пор, пока блокировка не будет снята. Этот вид блокировки обычно сводит к минимуму промежуток времени между снятием блокировки одним потоком и получением другим. Но это вычислительно затратно, потому что процессор тратит все свое время на проверку доступности блокировки, когда какой-то другой поток может выполнять продуктивную работу. Как правило, мьютексы требуют атомарных инструкций, таких как сравнение и замена, выборка и добавление или проверка и установка, чтобы они могли проверять и получать блокировку за одну операцию.
>
> Современные ОС, например Windows, предлагают более эффективные альтернативы спин-блокировкам. Например, мьютексы, основанные на асинхронных вызовах процедур, позволяют потокам ожидать мьютекс и переходить в состояние ожидания. Как только мьютекс становится доступным, операционная система пробуждает ожидающий поток и передает владение мьютексом. Это позволяет другим потокам выполнять продуктивную работу с процессором, который в противном случае был бы занят в спин-блокировке.
>
> В общем, не нужно беспокоиться о деталях того, как мьютексы реализуются операционной системой, если только они не станут узким местом в программе.

Для краткости этот раздел посвящен `lock_guard`. В листинге 19.12 показано, как выполнить рефакторинг листинга 19.11, чтобы использовать `lock_guard` вместо ручных манипуляций с мьютексом.

Листинг 19.12. Рефакторинг листинга 19.11 для использования lock_guard

```
#include <future>
#include <iostream>
#include <mutex>

using namespace std;

void goat_rodeo() {
  const size_t iterations{ 1'000'000 };
  int tin_cans_available{};
  mutex tin_can_mutex;
  auto eat_cans = async(launch::async, [&] {
    for(size_t i{}; i<iterations; i++) {
      lock_guard<mutex> guard{ tin_can_mutex }; ❶
      tin_cans_available--;
    }
```

```
  });
  auto deposit_cans = async(launch::async, [&] {
    for(size_t i{}; i<iterations; i++) {
      lock_guard<mutex> guard{ tin_can_mutex }; ❷
      tin_cans_available++;
    }
  });
  eat_cans.get();
  deposit_cans.get();
  cout << "Tin cans: " << tin_cans_available << "\n";
}

int main() {
  goat_rodeo();
  goat_rodeo();
  goat_rodeo();
}
--------------------------------------------------------------------------------
Tin cans: 0
Tin cans: 0
Tin cans: 0
```

Вместо того чтобы использовать блокировку и разблокировку для управления взаимным исключением, создается блокировка `lock_guard` в начале каждой области, где требуется синхронизация ❶ ❷. Поскольку механизм взаимного исключения является мьютексом, он указывается в качестве параметра шаблона `lock_guard`. Листинги 19.11 и 19.12 имеют эквивалентное поведение во время их выполнения. Объекты RAII не требуют каких-либо дополнительных затрат времени выполнения по сравнению с разблокировками программ и извлечениями вручную.

К сожалению, блокировки взаимного исключения связаны с затратами времени выполнения. Вы также могли заметить, что выполнение листингов 19.11 и 19.12 заняло значительно больше времени, чем листинга 19.10. Причина в том, что извлечение и разблокировка замков — относительно затратная операция. В листингах 19.11 и 19.12 `tin_can_mutex` извлекается, а затем разблокируется два миллиона раз. По сравнению с увеличением или уменьшением целого числа получение или снятие блокировки занимает значительно больше времени, поэтому использование мьютекса для синхронизации асинхронных задач является неоптимальным. В определенных ситуациях можно использовать потенциально более эффективный подход с применением атомарности.

ПРИМЕЧАНИЕ

Для получения дополнительной информации об асинхронных задачах и будущих проектах см. [futures.async].

Атомарные операции

Слово *атомарный* происходит от греческого *átomos*, что означает «неделимый». Операция является атомарной, если она происходит в неделимой единице. Другой

поток не может наблюдать за операцией на полпути. При введении в листинг 19.10 блокировки для создания листинга 19.11 вы сделали атомарные операции увеличения и уменьшения, потому что асинхронные задачи больше не могли чередовать операции чтения и записи в `tin_cans_available`. Как вы уже испытали при запуске этого решения на основе блокировок, этот подход очень медленный, потому что получение блокировок весьма затратно.

Другой подход заключается в использовании шаблона класса `std::atomic` в заголовке `<atomic>`, который предоставляет примитивы, часто используемые в *параллельном программировании без блокировок*. Параллельное программирование без блокировок решает проблемы гонки данных без использования блокировок. На многих современных архитектурах процессоры поддерживают атомарные инструкции. Используя атомарность, вы можете избежать блокировок, опираясь на атомарные аппаратные инструкции.

В этой главе не обсуждается `std::atomic` или то, как разрабатывать пользовательские решения без блокировок, потому что это невероятно сложно сделать правильно и лучше оставить на усмотрение экспертов. Однако в простых ситуациях (листинг 19.10), можно использовать `std::atomic`, чтобы убедиться, что операции увеличения или уменьшения не могут быть разделены. Это искусно решает проблему гонки данных.

Шаблон `std::atomic` предлагает специализации для всех основных типов, как показано в таблице 19.2.

Таблица 19.2. Спецификации шаблонов `std::atomic` для фундаментальных типов

Спецификация шаблона	Псевдоним
`std::atomic<bool>`	`std::atomic_bool`
`std::atomic<char>`	`std::atomic_char`
`std::atomic<unsigned char>`	`std::atomic_uchar`
`std::atomic<short>`	`std::atomic_short`
`std::atomic<unsigned short>`	`std::atomic_ushort`
`std::atomic<int>`	`std::atomic_int`
`std::atomic<unsigned int>`	`std::atomic_uint`
`std::atomic<long>`	`std::atomic_long`
`std::atomic<unsigned long>`	`std::atomic_ulong`
`std::atomic<long long>`	`std::atomic_llong`
`std::atomic<unsigned long long>`	`std::atomic_ullong`
`std::atomic<char16_t>`	`std::atomic_char16_t`
`std::atomic<char32_t>`	`std::atomic_char32_t`
`std::atomic<wchar_t>`	`std::atomic_wchar_t`

В таблице 19.3 перечислены некоторые из поддерживаемых операций для `std::atomic`. Шаблон `std::atomic` не имеет конструктора копирования.

Таблица 19.3. Поддерживаемые операции для `std::atomic`

Операция	Описание
`a{}` `a{ 123 }`	Конструктор по умолчанию. Инициализирует значение до `123`
`a.is_lock_free()`	Возвращает `true`, если `a` не имеет блокировки (зависит от ЦПУ)
`a.store(123)`	Сохраняет значение `123` в `a`
`a.load()` `a()`	Возвращает сохраненное значение
`a.exchange(123)`	Заменяет текущее значение на `123` и возвращает старое значение. Это операция «чтение-изменение-запись»
`a.compare_exchange_weak (10, 20)` `a.compare_exchange_strong (10, 20)`	Если текущее значение равно `10`, заменяется на `20`. Возвращает `true`, если значение было заменено. Смотрите [atomic] для подробностей о `weak` и `strong`

Для числовых типов специализации предлагаются дополнительные операции, перечисленные в таблице 19.4.

Таблица 19.4. Поддерживаемые операции для числовых специализаций `std::atomic a`

Операция	Описание
`a.fetch_add(123)` `a+=123`	Заменяет текущее значение на результат добавления аргумента к текущему значению. Возвращает значение до модификации. Это операция «чтение-изменение-запись»
`a.fetch_sub(123)` `a-=123`	Заменяет текущее значение на результат вычитания аргумента из текущего значения. Возвращает значение до модификации. Это операция «чтение-изменение-запись»
`a.fetch_and(123)` `a&=123`	Заменяет текущее значение результатом побитового `AND`, а аргумент — текущим значением. Возвращает значение до модификации. Это операция «чтение-изменение-запись»
`a.fetch_or(123)` `a|=123`	Заменяет текущее значение на результат побитового ИЛИ аргумента с текущим значением. Возвращает значение до модификации. Это операция «чтение-изменение-запись»
`a.fetch_xor(123)` `a^=123`	Заменяет текущее значение на результат побитового `XOR`, аргументируя текущее значение. Возвращает значение до модификации. Это операция «чтение-изменение-запись»
`a++` `a--`	Увеличивает или уменьшает `a`

ПРИМЕЧАНИЕ

Специализации для типов в `<cstdint>` также доступны. См. [atomics.syn] для дополнительной информации.

Поскольку листинг 19.12 является главным кандидатом для решения без блокировки, можно заменить тип `tin_cans_available` на `atomic_int` и удалить `mutex`. Это предотвращает состояние гонки, как показано в таблице 19.1.

Листинг 19.13 реализует этот рефакторинг.

Листинг 19.13. Разрешение состояния гонки с использованием atomic_int вместо mutex

```
#include <future>
#include <iostream>
#include <atomic>

using namespace std;

void goat_rodeo() {
  const size_t iterations{ 1'000'000 };
  atomic_int❶ tin_cans_available{};
  auto eat_cans = async(launch::async, [&] {
    for(size_t i{}; i<iterations; i++)
      tin_cans_available--; ❷
  });
  auto deposit_cans = async(launch::async, [&] {
    for(size_t i{}; i<iterations; i++)
      tin_cans_available++; ❸
  });
  eat_cans.get();
  deposit_cans.get();
  cout << "Tin cans: " << tin_cans_available << "\n";
}

int main() {
  goat_rodeo();
  goat_rodeo();
  goat_rodeo();
}
--------------------------------------------------------------------------------
Tin cans: 0
Tin cans: 0
Tin cans: 0
```

`int` заменяется на `atomic_int` ❶, а `mutex` удаляется. Поскольку операторы уменьшения ❷ и увеличения ❸ являются атомарными, состояние гонки остается решенным.

ПРИМЕЧАНИЕ

Для получения дополнительной информации об атомарности обратитесь к [atomics].

Вы также, вероятно, заметили значительное повышение производительности от листинга 19.12 до 19.13. В целом использование атомарных операций будет намного быстрее, чем получение мьютекса.

ПРЕДУПРЕЖДЕНИЕ

Если у вас нет очень простой проблемы одновременного доступа, такой как проблема в этом разделе, не стоит пытаться реализовать решение без блокировок самостоятельно. Обратитесь к библиотеке Boost Lockfree для получения высококачественных, тщательно протестированных контейнеров без блокировки. И как всегда, нужно решить, будет ли оптимальной реализация на основе блокировки или без блокировки.

Переменные условия

Переменная условия — это примитив синхронизации, который блокирует один или несколько потоков до получения уведомления. Другой поток может уведомить переменную условия. После уведомления переменная условия может разблокировать один или несколько потоков, чтобы они могли прогрессировать. Очень популярный шаблон условных переменных включает в себя поток, выполняющий следующие действия:

1. Получить мьютекс, совместно используемый с ожидающими потоками.
2. Изменить общее состояние.
3. Уведомить переменную условия.
4. Освободить мьютекс.

Все потоки, ожидающие переменную условия, затем выполняют следующие действия:

1. Получить мьютекс.
2. Дождаться условной переменной (это освободит мьютекс).
3. Когда другой поток уведомляет переменную условия, этот поток просыпается и может выполнять некоторую работу (это автоматически запрашивает мьютекс).
4. Освободить мьютекс.

Из-за запутанности, возникающей из-за сложности современных операционных систем, иногда потоки могут внезапно просыпаться. Поэтому важно убедиться, что условная переменная действительно была изменена после пробуждения ожидающего потока.

stdlib предоставляет `std::condition_variable` в заголовке `<condition_variable>`, который поддерживает несколько операций, в том числе перечисленные в таблице 19.5. `condition_variable`, поддерживает только конструкцию по умолчанию, а конструктор копирования удаляется.

Таблица 19.5. Поддерживаемые операции `std::condition_variable` cv

Операция	Описание
`cv.notify_one()`	Если какие-либо потоки ожидают на `cv`, эта операция уведомляет один из них
`cv.notify_all()`	Если какие-либо потоки ожидают на `cv`, эта операция уведомляет их все
`cv.wait(lock, [pred])`	С учетом блокировки мьютекса, принадлежащего уведомителю, возвращается при пробуждении. Если указано, `pred` определяет, является ли уведомление ложным (возвращает `false`) или реальным (возвращает `true`)
`cv.wait_for(lock, [durn], [pred])`	То же, что и `cv.wait`, за исключением `wait_for`, ожидающего только `durn`. Если происходит тайм-аут и `pred` не указан, возвращается `std::cv_status::timeout`; в противном случае возвращается `std::cv_status::no_timeout`
`cv.wait_until(lock, [time], [pred])`	То же, что и `wait_for`, за исключением того, что вместо `std::chrono::duration` используется `std::chrono::time_point`

Например, можно выполнить рефакторинг листинга 19.12, чтобы задача `deposit_cans` была завершена до выполнения задачи `eat_cans` с использованием переменной условия, как показано в листинге 19.14.

Листинг 19.14. Использование переменных условия для обеспечения хранения (deposit_cans) всех банок до их употребления (eat_cans)

```
#include <future>
#include <iostream>
#include <mutex>
#include <condition_variable>

using namespace std;

void goat_rodeo() {
  mutex m; ❶
  condition_variable cv; ❷
  const size_t iterations{ 1'000'000 };
  int tin_cans_available{};

  auto eat_cans = async(launch::async, [&] {
    unique_lock<mutex> lock{ m }; ❸
    cv.wait(lock, [&] { return tin_cans_available == 1'000'000; }); ❹
    for(size_t i{}; i<iterations; i++)
      tin_cans_available--;
  });

  auto deposit_cans = async(launch::async, [&] {
    scoped_lock<mutex> lock{ m }; ❺
```

```
    for(size_t i{}; i<iterations; i++)
      tin_cans_available++;
    cv.notify_all(); ❻
  });
  eat_cans.get();
  deposit_cans.get();
  cout << "Tin cans: " << tin_cans_available << "\n";
}

int main() {
  goat_rodeo();
  goat_rodeo();
  goat_rodeo();
}
--------------------------------------------------------------------------------
Tin cans: 0
Tin cans: 0
Tin cans: 0
```

Здесь объявляется `mutex` ❶ и `condition_variable` ❷, которые будут использоваться для координации асинхронных задач. В задаче `eat_cans` вы получаете `mutex` `unique_lock`, который передается в ожидание вместе с предикатом, который возвращает `true`, если есть доступные банки ❸. Этот метод освобождает мьютекс и затем блокируется до тех пор, пока не будут выполнены два условия: переменная `condition_variable` пробуждает этот поток, и один миллион банок становится доступным ❹ (напомним: нужно проверить, что все банки доступны из-за ложных пробуждений). В задаче `deposit_cans` код получает блокировку `mutex` ❺, вызывает соответствующий метод и затем уведомляет все потоки, заблокированные в условии `condition_variable` ❻.

Обратите внимание, что в отличие от всех предыдущих подходов, `tin_cans_available` не может быть отрицательным, потому что порядок хранения банок для хранения и употребления гарантирован.

ПРИМЕЧАНИЕ

Для получения дополнительной информации об условных переменных обратитесь к [thread.condition].

Низкоуровневые средства конкурентного программирования

Библиотека `<thread>` в stdlib содержит низкоуровневые средства для конкурентного программирования. Например, класс `std::thread` моделирует поток операционной системы. Однако лучше не использовать потоки напрямую, а создавать конкурентность в своих программах с абстракциями более высокого уровня — задачами. Если требуется низкоуровневый доступ к потокам, то [thread] предлагает больше информации.

Но библиотека <thread> содержит несколько полезных функций для управления текущим потоком.

- Функция std::this_thread::yield не принимает аргументов и возвращает void. Точное поведение yield зависит от среды, но в целом она дает подсказку, что операционная система должна дать возможность другим потокам работать. Это полезно, когда, например, существует высокая конкуренция за блокировку определенного ресурса и нужно, чтобы все потоки получили возможность доступа.
- Функция std::this_thread::get_id не принимает аргументов и возвращает объект типа std::thread::id, который является облегченным потоком, поддерживающим операторы сравнения и operator<<. Как правило, он используется в качестве ключа в ассоциативных контейнерах.
- Функция std::this_thread::sleep_for принимает аргумент std::chrono::duration, блокирует выполнение в текущем потоке до тех пор, пока не пройдет хотя бы указанный период времени, и возвращает void.
- Функция std::this_thread::sleep_until принимает std::chrono::time_point и возвращает void. Она полностью аналогична sleep_for за исключением того, что она блокирует поток по крайней мере до указанной точки времени.

Когда эти функции нужны, они просто незаменимы. В противном случае не стоит взаимодействовать с заголовком <thread>.

Параллельные алгоритмы

В главе 18 были рассмотрены алгоритмы stdlib, многие из которых принимают необязательный первый аргумент, называемый его политикой выполнения, закодированной значением std::execute. В поддерживаемых средах возможны три значения: seq, par и par_unseq. Последние два параметра указывают, что вы хотите выполнить алгоритм параллельно.

Пример: параллельная сортировка

В листинге 19.15 показано, как изменение одного аргумента с seq на par может оказать огромное влияние на время выполнения программы, отсортировав миллиарды чисел в обоих направлениях.

Листинг 19.15. Сортировка миллиарда чисел с использованием std::sort с std::execute::seq против std::execute::par. (Результаты получены на компьютере с Windows 10 x64 с двумя процессорами Intel XeonE5-2620 v3.)

```
#include <algorithm>
#include <vector>
#include <numeric>
#include <random>
#include <chrono>
```

```
#include <iostream>
#include <execution>

using namespace std;

// Из листинга 12.25:
struct Stopwatch {
--пропуск--
};

vector<long> make_random_vector() { ❶
  vector<long> numbers(1'000'000'000);
  iota(numbers.begin(), numbers.end(), 0);
  mt19937_64 urng{ 121216 };
  shuffle(numbers.begin(), numbers.end(), urng);
  return numbers;
}

int main() {
  cout << "Constructing random vectors...";
  auto numbers_a = make_random_vector(); ❷
  auto numbers_b{ numbers_a }; ❸
  chrono::nanoseconds time_to_sort;
  cout << " " << numbers_a.size() << " elements.\n";
  cout << "Sorting with execution::seq...";
  {
    Stopwatch stopwatch{ time_to_sort };
    sort(execution::seq, numbers_a.begin(), numbers_a.end()); ❹
  }
  cout << " took " << time_to_sort.count() / 1.0E9 << " sec.\n";

  cout << "Sorting with execution::par...";
  {
    Stopwatch stopwatch{ time_to_sort };
    sort(execution::par, numbers_b.begin(), numbers_b.end()); ❺
  }
  cout << " took " << time_to_sort.count() / 1.0E9 << " sec.\n";
}
--------------------------------------------------------------------------------
Constructing random vectors... 1000000000 elements.
Sorting with execution::seq... took 150.489 sec.
Sorting with execution::par... took 17.7305 sec.
```

Функция `make_random_vector` ❶ создает вектор, содержащий миллиард уникальных чисел. Создаются две копии: `numbers_a` ❷ и `numbers_b` ❸. Каждый вектор сортируется отдельно. В первом случае сортировка выполняется с помощью политики последовательного выполнения ❹, а секундомер указывает, что операция заняла около двух с половиной минут (около 150 секунд). Во втором случае сортировка проводится с политикой параллельного выполнения ❺. На этот раз `Stopwatch` указывает, что операция заняла около 18 секунд. Последовательное выполнение заняло примерно в 8,5 раза больше времени.

Параллельные алгоритмы — это не магия

К сожалению, параллельные алгоритмы не являются магическими. Хотя они отлично работают в простых ситуациях, таких как `sort` в листинге 19.15, нужно быть осторожным при их использовании. Каждый раз, когда алгоритм производит побочные эффекты за пределами целевой последовательности, стоит тщательно продумывать состояние гонки. Красный флаг — это любой алгоритм, который передает объект функции в алгоритм. Если функциональный объект имеет совместное изменяемое состояние, выполняющиеся потоки будут иметь общий доступ и может возникнуть состояние гонки. Например, рассмотрим в листинге 19.16 параллельный вызов `transform`.

Листинг 19.16. Программа, содержащая состояние гонки из-за неатомарного доступа к n_transformed

```
#include <algorithm>
#include <vector>
#include <iostream>
#include <numeric>
#include <execution>

int main() {
  std::vector<long> numbers{ 1'000'000 }, squares{ 1'000'000 }; ❶
  std::iota(numbers.begin(), numbers.end(), 0); ❷
  size_t n_transformed{}; ❸
  std::transform(std::execution::par, numbers.begin(), numbers.end(), ❹
                 squares.begin(), [&n_transformed] (const auto x) {
                  ++n_transformed; ❺
                  return x * x; ❻
                 });
  std::cout << "n_transformed: " << n_transformed << std::endl; ❼
}
---------------------------------------------------------------------------
n_transformed: 187215 ❼
```

Все начинается с инициализации двух объектов `vector`, `numbers` и `squares`, которые содержат миллион элементов ❶. Затем один из них заполняется числами, используя `iota` ❷, и инициализируется переменная `n_transformed` со значением 0 ❸. Затем вызывается преобразование с политикой параллельного выполнения, `numbers` в качестве целевой последовательности, `squares` в качестве последовательности результатов и простым лямбда-выражением ❹. Лямбда-выражение увеличивает `n_transformed` ❺ и возвращает квадрат аргумента `x` ❻. Поскольку несколько потоков выполняют это лямбда-выражение, доступ к `n_transformed` должен быть синхронизирован ❼.

В предыдущем разделе были представлены два способа решения этой проблемы: блокировки и атомарность. В этом сценарии, вероятно, лучше всего использовать `std::atomic_size_t` в качестве замены для `size_t`.

Итоги

В этой главе рассматриваются конкурентность и параллелизм на очень высоком уровне. Кроме того, вы узнали, как запускать асинхронные задачи, которые позволяют легко внедрять концепции конкурентного программирования в код. Хотя введение параллельных и конкурентных концепций в программы может значительно повысить производительность, стоит тщательно избегать введения состояния гонки, которые допускают неопределенное поведение. Вы также узнали несколько механизмов для синхронизации доступа к изменяемым общим состояниям: мьютексы, условные переменные и атомарность.

Упражнения

19.1. Напишите свой собственный мьютекс на основе спин-блокировки, который называется `SpinLock`. Выставьте блокировку, `try_lock` и метод разблокировки. Ваш класс должен удалить конструктор копирования. Попробуйте использовать `std::lock_guard<SpinLock>` с экземпляром вашего класса.

19.2. Прочитайте о печально известной схеме блокировки с двойной проверкой (DCLP) и о том, почему вы не должны ее использовать. (См. статью Скотта Мейерса и Андрея Александреску, упомянутую в разделе «Что еще почитать».) Затем прочитайте о соответствующем способе обеспечения того, чтобы вызываемый объект вызывался ровно один раз с использованием `std::call_once` в [thread.once.callonce].

19.3. Создайте потокобезопасный класс очереди. Этот класс должен предоставлять интерфейс, такой как `std::queue` (см. [queue.defn]). Под капотом используйте `std::queue` для хранения элементов. Используйте `std::mutex` для синхронизации доступа к этой внутренней `std::queue`.

19.4. Добавьте метод `wait_and_pop` и член `std::condition_variable` в вашу потокобезопасную очередь. Когда пользователь вызывает `wait_and_pop` и очередь содержит элемент, она должна вытолкнуть элемент из очереди и вернуть его. Если очередь пуста, поток должен блокироваться до тех пор, пока элемент не станет доступным, а затем приступить к извлечению элемента.

19.5. (Необязательно) Прочитайте документацию Boost Coroutine2, особенно разделы «Overview», «Introduction» и «Motivation».

Что еще почитать?

- «C++ and The Perils of Double-Checked Locking: Part I», Scott Meyers, Andrei Alexandrescu (www.drdobbs.com/cpp/c-and-the-perils -of-double-check-locki/184405726 /)
- Международный стандарт ИСО/МЭК (2017) — Язык программирования C++ (Международная организация по стандартизации; Женева, Швейцария; isocpp.org/std/the-standard/)
- «Параллельное программирование на C++ в действии: практика разработки конкурентных программ», Энтони Уильямс (ДМК Пресс, 2016)
- «Effective Concurrency: Know When to Use an Active Object Instead of a Mutex», Herb Sutter (herbsutter.com/2010/09/24/effective-concurrency-know-when-to-use-an-active-object-instead-of-a-mutex/)
- «Эффективный и современный C++. 42 рекомендации по использованию C++11 и C++14», Скотт Мейерс (Вильямс, 2019)
- «A Survey of Modern Integer Factorization Algorithms», Peter L. Montgomery. CWI Quarterly 7.4 (1994): 337–365.

20 Сетевое программирование с помощью Boost Asio

Любой, кто потерял счет времени при использовании компьютера, знает склонность к мечте, желание воплощать мечты в жизнь и склонность пропускать обед.

Тим Бернерс-Ли

Boost Asio — это библиотека для низкоуровневого программирования ввода/вывода. В этой главе вы узнаете об основных сетевых возможностях Boost Asio, которые позволяют программам легко и эффективно взаимодействовать с сетевыми ресурсами. К сожалению, stdlib не содержит библиотеку сетевого программирования на C++17. По этой причине Boost Asio играет центральную роль во многих программах на C++ с сетевым компонентом.

Хотя Boost Asio является основным выбором для разработчиков на C++, которые хотят включить в свои программы кросс-платформенный высокопроизводительный ввод-вывод, общеизвестно, что это сложная библиотека. Это осложнение вкупе с незнанием низкоуровневого сетевого программирования может сбивать с толку новичков. Если эта глава кажется непонятной или информация о сетевом программировании вам не нужна, смело пропускайте.

ПРИМЕЧАНИЕ

Boost Asio также содержит средства для ввода-вывода с последовательными портами, потоками и некоторыми объектами, специфичными для операционной системы. Ее название происходит от фразы «асинхронный ввод-вывод». См. документацию Boost Asio для получения дополнительной информации.

Модель программирования Boost Asio

В модели программирования Boost *объект контекста ввода/вывода* абстрагирует интерфейсы операционной системы, которые обеспечивают асинхронную обработку данных. Этот объект является реестром для *объектов ввода-вывода*, которые инициируют асинхронные операции. Каждый объект знает свой соответствующий сервис, а объект контекста опосредует соединение.

ПРИМЕЧАНИЕ

Все классы Boost Asio отображаются в служебном заголовке `<boost/asio.hpp>`.

Boost Asio определяет один объект сервиса, `boost::asio::io_context`. Его конструктор принимает необязательный целочисленный аргумент, называемый подсказкой конкурентности и представляющий собой число потоков, которые `io_context` должен разрешить для одновременной работы. Например, на восьмиядерном компьютере можно создать `io_context` следующим образом:

```
boost::asio::io_context io_context{ 8 };
```

Тот же объект `io_context` передается в конструкторы объектов ввода/вывода. После настройки всех объектов ввода-вывода вызовется метод `run` для `io_context`, который будет блокироваться до завершения всех ожидающих операций ввода-вывода.

Одним из самых простых объектов ввода-вывода является `boost::asio::stable_timer`, который можно использовать для планирования задач. Его конструктор принимает объект `io_context` и необязательный `std::chrono::time_point` или `std::chrono_duration`. Например, следующий код создаст `steady_timer`, срок действия которого истекает через три секунды:

```
boost::asio::steady_timer timer{
  io_context, std::chrono::steady_clock::now() + std::chrono::seconds{ 3 }
};
```

Можно подождать по таймеру с блокирующим или неблокирующим вызовом. Чтобы заблокировать текущий поток, используется метод `wait`. Результат по сути аналогичен использованию `std::this_thread::sleep_for`, о котором вы узнали в разделе «Chrono» на с. 463. Для асинхронного ожидания используется метод `async_wait` таймера. Он принимает функциональный объект, называемый *обратным вызовом*. Операционная система вызовет функциональный объект, как только наступит время для пробуждения потока. Из-за сложностей, связанных с современными операционными системами, это может происходить или не происходить из-за истечения срока действия таймера.

По истечении времени таймера можно создать другой таймер, если требуется выполнить дополнительное ожидание. Если вызвать `wait` для истекшего таймера, он немедленно завершит работу. Это, вероятно, не то, что вы намереваетесь сделать, поэтому убедитесь, что `wait` вызывается на не истекших таймерах.

Чтобы проверить, истек ли таймер, функциональный объект должен принять `boost::system::error_code`. Класс `error_code` — это простой класс, представляющий ошибки, специфичные для операционной системы. Он неявно преобразуется в `bool` (`true`, если представляет условие ошибки; в противном случае — `false`). Если код ошибки обратного вызова оценивается как `false`, таймер истек.

После того как асинхронная операция ставится в очередь с использованием `async_wait`, вызывается метод `run` для объекта `io_context`, потому что этот метод блокируется до завершения всех асинхронных операций.

В листинге 20.1 показано, как создать и использовать таймеры для блокирующих и неблокирующих ожиданий.

Листинг 20.1. Программа, использующая boost::asio::stable_timer для синхронного и асинхронного ожидания

```
#include <iostream>
#include <boost/asio.hpp>
#include <chrono>

boost::asio::steady_timer make_timer(boost::asio::io_context& io_context) { ❶
  return boost::asio::steady_timer{
          io_context,
          std::chrono::steady_clock::now() + std::chrono::seconds{ 3 }
  };
}

int main() {
  boost::asio::io_context io_context; ❷

  auto timer1 = make_timer(io_context); ❸
  std::cout << "entering steady_timer::wait\n";
  timer1.wait(); ❹
  std::cout << "exited steady_timer::wait\n";

  auto timer2 = make_timer(io_context); ❺
  std::cout << "entering steady_timer::async_wait\n";
  timer2.async_wait([] (const boost::system::error_code& error) { ❻
    if (!error) std::cout << "<<callback function>>\n";
  });
  std::cout << "exited steady_timer::async_wait\n";
  std::cout << "entering io_context::run\n";
  io_context.run(); ❼
  std::cout << "exited io_context::run\n";
}
--------------------------------------------------------------------------------
entering steady_timer::wait
exited steady_timer::wait
entering steady_timer::async_wait
exited steady_timer::async_wait
entering io_context::run
<<callback function>>
exited io_context::run
```

Определяется функция `make_timer` для создания `steady_timer`, срок действия которого истекает через три секунды ❶. В `main` инициализируется `io_context` ❷ программы и создается первый таймер из `make_timer` ❸. При вызове `wait` для этого таймера ❹ поток блокируется на три секунды, прежде чем продолжить. Затем создается другой таймер с помощью `make_timer` ❺, а затем вызывается `async_wait` с лямбда-выражением, которое выводит `<<callback_function>>`, когда время таймера истекает ❻. Наконец, запускается `run` для `io_context`, чтобы начать обработку операций ❼.

Сетевое программирование с помощью Asio

Boost Asio содержит средства для выполнения сетевого ввода-вывода через несколько важных сетевых протоколов. Теперь, когда вы знаете основы использования `io_context` и то, как ставить в очередь асинхронные операции ввода/вывода, можно изучить, как выполнять более сложные виды ввода/вывода. В этом разделе вы расширите знания об ожидании таймеров и научитесь использовать возможности сетевого ввода-вывода Boost Asio. К концу этой главы вы узнаете, как создавать программы, которые обмениваются данными по сети.

Модуль интернет-протокола

Интернет-протокол (IP) является основным протоколом для передачи данных по сетям. Каждый участник в IP-сети называется *хостом*, и каждый хост получает IP-адрес для его идентификации. IP-адреса бывают двух версий: IPv4 и IPv6. Адрес IPv4 составляет 32 бита, а адрес IPv6 — 128 бит.

Протокол управляющих сообщений интернета (ICMP) используется сетевыми устройствами для отправки информации, поддерживающей работу IP-сети. Программы ping и traceroute используют сообщения ICMP для запроса сети. Как правило, приложениям конечного пользователя не требуется напрямую взаимодействовать с ICMP.

Для отправки данных по IP-сети обычно используется либо протокол управления передачей (TCP), либо протокол пользовательских дейтаграмм (UDP).

В общем, TCP используется, когда нужно быть уверенным, что данные прибывают в место назначения, а UDP — когда нужно быть уверенным, что данные проходят быстро. TCP — это протокол, ориентированный на установление соединения, в котором получатели подтверждают, что они получили предназначенные для них сообщения. UDP — это простой протокол без установления соединения, который не имеет встроенной надежности.

С TCP и UDP сетевые устройства соединяются друг с другом через порты. Порт — это целое число в диапазоне от 0 до 65 535 (2 байта), которое указывает конкретную службу, работающую на данном сетевом устройстве. Таким образом, на одном устройстве может работать несколько служб, и к каждой из них можно обращаться

отдельно. Когда одно устройство, называемое *клиентом*, инициирует связь с другим устройством, называемым сервером, клиент указывает, к какому порту он хочет подключиться. При связи IP-адреса устройства с номером порта результат называется *сокетом*.

Например, устройство с IP-адресом 10.10.10.100 может обслуживать веб-страницу, привязав приложение веб-сервера к порту 80. Это создает сокет сервера в 10.10.10.100:80. Затем устройство с IP-адресом 10.10.10.200 запускает веб-браузер, который открывает «случайный высокий порт», такой как 55123. Это создает клиентский сокет в 10.10.10.200:55123. Затем клиент подключается к серверу, создавая TCP-соединение между сокетом клиента и сокетом сервера. Многие другие процессы могут выполняться на одном или обоих устройствах одновременно со многими другими сетевыми подключениями.

Управление по присвоению номеров в интернете (IANA) ведет список назначенных номеров, чтобы стандартизировать порты, которые пользуются определенными видами услуг (список доступен по адресу www.iana.org). Таблица 20.1 содержит несколько часто используемых протоколов в этом списке.

Таблица 20.1. Известные протоколы, назначенные IANA

Порт	TCP	UDP	Ключевое слово	Описание
7	✓	✓	`echo`	Эхо-протокол
13	✓	✓	`daytime`	Дневной протокол
21	✓		`ftp`	Протокол передачи файлов
22	✓		`ssh`	Протокол безопасной оболочки
23	✓		`telnet`	Протокол Telnet
25	✓		`smtp`	Протокол передачи простых сообщений
53	✓	✓	`domain`	Система доменных имен
80	✓		`http`	Протокол передачи гипертекста
110	✓		`pop3`	Протокол почтового отделения
123		✓	`ntp`	Протокол сетевого времени
143	✓		`imap`	Протокол доступа к сообщениям в интернете
179	✓		`bgp`	Протокол пограничной маршрутизации
194	✓		`irc`	Ретранслируемый интернет-чат
443	✓		`https`	Протокол передачи гипертекста (безопасный)

Boost Asio поддерживает сетевой ввод-вывод через ICMP, TCP и UDP. Для краткости в этой главе обсуждается только TCP, поскольку классы Asio, задействованные во всех трех протоколах, очень похожи.

ПРИМЕЧАНИЕ

Вам может быть интересно узнать, что означает соединение в контексте TCP/UDP, или протокол без установления соединения может показаться абсурдным. Здесь соединение означает установление канала между двумя участниками сети, который гарантирует доставку и сохранение порядка сообщений. Эти участники выполняют синхронизацию для установления соединения, и у них есть механизм информирования друг друга о том, что они хотят закрыть соединение. В протоколе без установления соединения участник отправляет пакет другому участнику без предварительного установления канала.

ПРИМЕЧАНИЕ

Если вы не знакомы с сетевыми протоколами, советую почитать книгу «The TCP/IP Guide» Чарльза Козиерока (Charles M. Kozierok).

Разрешение имени хост-системы

Когда клиент хочет подключиться к серверу, ему нужен IP-адрес сервера. Иногда он уже может иметь эту информацию. В других случаях клиент может иметь только имя службы. Процесс преобразования имени службы в IP-адрес называется *разрешением имени хоста*. Boost Asio содержит класс `boost::asio::ip::tcp::resolver` для выполнения разрешения имени хоста. Чтобы создать распознаватель, экземпляр `io_context` передается в качестве единственного параметра конструктора, как показано ниже:

```
boost::asio::ip::tcp::resolver my_resolver{ my_io_context };
```

Чтобы выполнить разрешение имени хоста, используется метод `resolve`, который принимает как минимум два аргумента `string_view`: имя хоста и сервис. Можно указать ключевое слово или номер порта для службы (см. таблицу 20.1 для некоторых примеров ключевых слов). Метод `resolve` возвращает диапазон объектов `boost::asio::ip::tcp::resolver::basic_resolver_entry`, которые предоставляют несколько полезных методов:

- `endpoint` получает IP-адрес и порт;
- `host_name` получает имя хоста;
- `service_name` получает имя службы, связанной с этим портом.

Если разрешение не удается, выдается `boost::system::system_error`. В качестве альтернативы можно передать ссылку `boost::system::error_code`, которая получает ошибку вместо создания исключения. Например, листинг 20.2 определяет IP-адрес и порт для веб-сервера No Starch Press с использованием Boost Asio.

ПРИМЕЧАНИЕ

Результаты могут отличаться в зависимости от того, где находятся веб-серверы No Starch Press в IP-пространстве.

Листинг 20.2. Блокировка разрешения имени хоста с помощью Boost Asio

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

int main() {
  boost::asio::io_context io_context; ❶
  boost::asio::ip::tcp::resolver resolver{ io_context }; ❷
  boost::system::error_code ec;
  for(auto&& result : resolver.resolve("www.nostarch.com", "http", ec)) { ❸
    std::cout << result.service_name() << " " ❹
              << result.host_name() << " " ❺
              << result.endpoint() ❻
              << std::endl;
  }
  if(ec) std::cout << "Error code: " << ec << std::endl; ❼
}
--------------------------------------------------------------------------------
http www.nostarch.com 104.20.209.3:80
http www.nostarch.com 104.20.208.3:80
```

В примере инициализируются `io_context` ❶ и `boost::asio::ip::tcp::resolver` ❷. В пределах цикла `for` на основе диапазона перебирается каждый `result` ❸ и извлекаются `service_name` ❹, `host_name` ❺ и `endpoint` ❻. Если `resolve` встречает ошибку, она выводится в стандартный вывод ❼.

Можно выполнить асинхронное разрешение имени хоста, используя метод `async_resolve`. Как и с `resolve`, имя хоста и служба передаются в качестве первых двух аргументов. Кроме того, предоставляется функциональный объект обратного вызова, который принимает два аргумента: `system_error_code` и диапазон объектов `basic_resolver_entry`. В листинге 20.3 показано, как выполнить рефакторинг в листинге 20.2, чтобы вместо него использовать асинхронное разрешение имени хоста.

Листинг 20.3. Рефакторинг листинга 20.2 для использования async_resolve

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

int main() {
  boost::asio::io_context io_context;
  boost::asio::ip::tcp::resolver resolver{ io_context };
  resolver.async_resolve("www.nostarch.com", "http", ❶
    [](boost::system::error_code ec, const auto& results) { ❷
      if (ec) { ❸
        std::cerr << "Error:" << ec << std::endl;
        return; ❹
      }
      for (auto&& result : results) { ❺
        std::cout << result.service_name() << " "
                  << result.host_name() << " "
                  << result.endpoint() << " "
                  << std::endl; ❻
      }
```

```
    }
  );
  io_context.run(); ❼
}
-------------------------------------------------------------------------------
http www.nostarch.com 104.20.209.3:80
http www.nostarch.com 104.20.208.3:80
```

Настройка идентична листингу 20.2 до тех пор, пока вы не вызовете `async_resolve` для преобразователя ❶. Передаются те же имя хоста и служба, что и раньше, но добавляется аргумент обратного вызова, который принимает обязательные параметры ❷. В теле лямбда-выражения обратного вызова проверяется условие ошибки ❸. Если оно существует, выводится понятное сообщение об ошибке и возвращается результат ❹. В безошибочном случае результаты перебираются, как и раньше ❺, выводя `service_name`, `host_name` и `endpoint` ❻. Как и в случае с таймером, нужно вызвать `run` для `io_context`, чтобы дать асинхронным операциям возможность завершиться ❼.

Соединение

Получив диапазон конечных точек либо с помощью разрешения имени хоста, либо создав его самостоятельно, можно установить соединение.

Во-первых, понадобится `boost::asio::ip::tcp::socket`, класс, который абстрагирует сокет базовой операционной системы и представляет его для использования в Asio. Сокет принимает `io_context` в качестве аргумента.

Во-вторых, нужно вызвать функцию `boost::asio::connect`, которая принимает `socket`, представляющий конечную точку, с которой нужно соединиться, в качестве первого аргумента и диапазон `endpoint` в качестве второго аргумента. Можно предоставить ссылку на код ошибки в качестве необязательного третьего аргумента; в противном случае `connect` вызовет исключение `system_error`, если произойдет ошибка. В случае успеха `connect` возвращает один `endpoint` — `endpoint` в диапазоне ввода, к которому он успешно подключился. После этой точки объект socket представляет собой настоящий сокет в среде системы.

В листинге 20.4 показано, как подключиться к веб-серверу No Starch Press.

Создается `resolver` ❶, как показано в листинге 20.3. Кроме того, инициализируется `socket` с тем же `io_context` ❷. Далее вызывается метод `resolve` для получения каждого `endpoint`, связанного с www.nostarch.com, через порт 80 ❸. Напомним, что каждый `endpoint` — это IP-адрес и порт, соответствующий разрешенному хосту. В этом случае для определения использовалась система доменных имен, чтобы определить, что www.nostarch.com на порте 80 находится по IP-адресу 104.20.209.3. Затем вызывается `connect`, используя свой сокет и конечные точки ❹, возвращающий конечную точку, с которой успешно установлено соединение ❺. В случае ошибки `resolve` или `connect` вызовут исключение, которое будет перехвачено и выведено в стандартный вывод ошибок ❻.

Листинг 20.4. Соединение с веб-сервером No Starch

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

int main() {
  boost::asio::io_context io_context;
  boost::asio::ip::tcp::resolver resolver{ io_context }; ❶
  boost::asio::ip::tcp::socket socket{ io_context }; ❷
  try {
    auto endpoints = resolver.resolve("www.nostarch.com", "http"); ❸
    const auto connected_endpoint = boost::asio::connect(socket, endpoints); ❹
    std::cout << connected_endpoint; ❺
  } catch(boost::system::system_error& se) {
    std::cerr << "Error: " << se.what() << std::endl; ❻
  }
}
--------------------------------------------------------------------------------
104.20.209.3:80 ❺
```

Также можно подключиться асинхронно с помощью `boost::asio::async_connect`, который принимает те же два аргумента, что и `connect`: сокет и диапазон `endpoint`. Третий аргумент — это функциональный объект, действующий как функция обратного вызова, которая должна принять код ошибки в качестве первого аргумента и конечную точку в качестве второго аргумента. Листинг 20.5 показывает, как соединиться асинхронно.

Листинг 20.5. Асинхронное соединение с веб-сервером No Starch

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

int main() {
  boost::asio::io_context io_context;
  boost::asio::ip::tcp::resolver resolver{ io_context };
  boost::asio::ip::tcp::socket socket{ io_context };
  boost::asio::async_connect(socket, ❶
    resolver.resolve("www.nostarch.com", "http"), ❷
    [] (boost::system::error_code ec, const auto& endpoint){ ❸
      std::cout << endpoint; ❹
  });
  io_context.run(); ❺
}
--------------------------------------------------------------------------------
104.20.209.3:80 ❹
```

Настройка точно такая же, как в листинге 20.4, за исключением того, что `connect` заменяется на `async_connect` и передаются одинаковые первый ❶ и второй ❷ аргументы. Третий аргумент — это функциональный объект ❸ обратного вызова, внутри которого конечная точка выводится в стандартный вывод ❹. Как и во всех асинхронных программах Asio, выполняется вызов для `io_context` ❺.

Буферы

Boost Asio предоставляет несколько буферных классов. *Буфер* (или *буфер данных*) — это память, в которой хранятся временные данные. Буферные классы Boost Asio формируют интерфейс для всех операций ввода-вывода. Прежде чем вы сможете что-либо делать с сетевыми подключениями, понадобится интерфейс для чтения и записи данных.

Для этого потребуются только три типа буфера:

- `boost::asio::const_buffer` содержит буфер, который нельзя изменить после создания;
- `boost::asio::mutable_buffer` содержит буфер, который можно изменить после создания;
- `boost::asio::streambuf` содержит автоматически изменяемый размер буфера на основе `std::streambuf`.

Все три буферных класса предоставляют два важных метода для доступа к их базовым данным: `data` и `size`.

Методы `data` классов `mutable_buffer` и `const_buffer` возвращают указатель на первый элемент в базовой последовательности данных, а их методы `size` возвращают количество элементов в этой последовательности. Элементы являются непрерывными. Оба буфера предоставляют конструкторы по умолчанию, которые инициализируют пустой буфер, как показано в листинге 20.6.

Листинг 20.6. Конструкторы по умолчанию const_buffer и mutable_buffer возвращают пустые буферы

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

TEST_CASE("const_buffer default constructor") {
  boost::asio::const_buffer cb; ❶
  REQUIRE(cb.size() == 0); ❷
}

TEST_CASE("mutable_buffer default constructor") {
  boost::asio::mutable_buffer mb; ❸
  REQUIRE(mb.size() == 0); ❹
}
```

При использовании конструкторов по умолчанию ❶ ❸ создаются пустые буферы с нулевым `size` ❷ ❹.

И `mutable_buffer`, и `const_buffer` предоставляют конструкторы, которые принимают `void*` и `size_t`, соответствующие данным, которые нужно обернуть. Обратите внимание, что эти конструкторы не берут на себя ответственность за указанную память, поэтому *нужно убедиться, что продолжительность хранения этой памяти, по крайней мере, равна времени существования создаваемого буфера.* Это проектное

решение, которое дает вам как пользователю Boost Asio максимальную гибкость. К сожалению, это также приводит к потенциально неприятным ошибкам. Неправильное управление временем жизни буферов и объектов, на которые они указывают, приведет к неопределенному поведению.

В листинге 20.7 показано, как создать буферы с помощью конструктора, основанного на указателе.

Листинг 20.7. Создание const_buffer и mutable_buffer с использованием конструктора на основе указателей

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

TEST_CASE("const_buffer constructor") {
  boost::asio::const_buffer cb{ "Blessed are the cheesemakers.", 7 }; ❶

  REQUIRE(cb.size() == 7); ❷
  REQUIRE(*static_cast<const char*>(cb.data()) == 'B'); ❸
}

TEST_CASE("mutable_buffer constructor") {
  std::string proposition{ "Charity for an ex-leper?" };
  boost::asio::mutable_buffer mb{ proposition.data(), proposition.size() }; ❹

  REQUIRE(mb.data() == proposition.data()); ❺
  REQUIRE(mb.size() == proposition.size()); ❻
}
```

В первом тесте создается `const_buffer`, используя строку в стиле C и фиксированную длину 7 ❶. Эта фиксированная длина меньше, чем длина строкового литерала `Blessed are the cheesemakers.`, поэтому этот буфер ссылается на `Blessed`, а не на всю строку. Это показывает, что можно выбрать подмножество массива (так же, как с `std::string_view`, о котором вы узнали в «Строковом представлении» на с. 586). Полученный буфер имеет размер 7 ❷, и если вы приведете указатель из данных к `const char*`, то увидите, что он указывает на символ `B` из строки в стиле C ❸.

Во втором тесте создается `mutable_buffer`, используя строку, вызывая его элементы `data` и `size` в конструкторе буфера ❹. Методы `data` ❺ и `size` ❻ полученного буфера возвращают идентичные данные в исходную строку.

Класс `boost::asio::streambuf` принимает два необязательных аргумента конструктора: максимальный размер `size_t` и распределитель. По умолчанию максимальный размер равен `std::numeric_limits <std::size_t>`, а распределитель аналогичен распределителю по умолчанию для контейнеров stdlib. Начальный размер входной последовательности `streambuf` всегда равен нулю, как показано в листинге 20.8.

`streambuf` ❶ создается по умолчанию, и при вызове метода `size`, возвращается 0 ❷.

Листинг 20.8. Создание streambuf по умолчанию

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

TEST_CASE("streambuf constructor") {
  boost::asio::streambuf sb; ❶
  REQUIRE(sb.size() == 0); ❷
}
```

Можно передать указатель на `streambuf` в конструктор `std::istream` или `std::ostream`. Вспомните из раздела «Классы потоков» на с. 613, что это специализации `basic_istream` и `basic_ostream`, которые предоставляют потоковые операции базовой синхронизации источнику. Листинг 20.9 показывает, как записывать и впоследствии читать из потокового буфера, используя эти классы.

Листинг 20.9. Запись в потоковый буфер и чтение из него

```
TEST_CASE("streambuf input/output") {
  boost::asio::streambuf sb; ❶
  std::ostream os{ &sb }; ❷
  os << "Welease Wodger!"; ❸

  std::istream is{ &sb }; ❹
  std::string command; ❺
  is >> command; ❻

  REQUIRE(command == "Welease"); ❼
}
```

Снова создается пустой `streambuf` ❶, и его адрес передается в конструктор `ostream` ❷. Затем записывается строка `WeleaseWodger!` в `ostream`, который, в свою очередь, записывает строку в основной поток `streambuf` ❸.

Затем снова создается `istream`, используя адрес `streambuf` ❹. Затем создается `string` ❺, и `istream` записывается в `string` ❻. Вспомните из «Специального форматирования для основных типов» на с. 619, что эта операция пропустит любой начальный пробел, а затем прочитает следующую строку до следующего пробела. Это дает первое слово строки, `Welease` ❼.

Boost Asio также предлагает шаблон вспомогательной функции `boost::asio::buffer`, который принимает `std::array` или `std::vector` элементов POD или `std::string`. Например, можно создать `mutable_buffer` с поддержкой `std::string` в листинге 20.7, используя вместо этого следующую конструкцию:

```
std::string proposition{ "Charity for an ex-leper?" };
auto mb = boost::asio::buffer(proposition);
```

Шаблон `buffer` является специализированным, поэтому, если предоставить аргумент `const`, он вернет `const_buffer`. Другими словами, чтобы сделать `const_buffer` из `proposition`, просто сделайте его `const`:

```
const std::string proposition{ "Charity for an ex-leper?" };
auto cb = boost::asio::buffer(proposition);
```

Вы создали `conb_buffer cb`.

Кроме того, можно создать динамический буфер, который представляет собой динамически изменяемый буфер, поддерживаемый `std::string` или `std::vector`. Можно создать его, используя шаблон функции `boost::asio::dynamic_buffer`, который принимает строку или вектор и возвращает `boost::asio::dynamic_string_buffer` или `boost::asio::dynamic_vector_buffer` в зависимости от ситуации. Например, можно создать динамический буфер, используя следующую конструкцию:

```
std::string proposition{ "Charity for an ex-leper?" };
auto db = boost::asio::dynamic_buffer(proposition);
```

Хотя динамический буфер имеет динамическое изменение размера, напомним, что классы `vector` и `string` используют распределитель, и это распределение может быть относительно медленной операцией. Итак, если вы знаете, сколько данных будет записано в буфер, можно добиться лучшей производительности, используя нединамический буфер. Как всегда, измерения и эксперименты помогут решить, какой подход выбрать.

Чтение и запись данных с помощью буферов

Обладая новыми знаниями о том, как хранить и извлекать данные с использованием буферов, вы можете научиться извлекать данные из сокета. Можно читать данные из активных объектов сокетов в буферные объекты, используя встроенные функции Boost Asio. Для блокировки чтения Boost Asio предлагает три функции:

- `boost::asio::read` пытается прочитать блок данных фиксированного размера;
- `boost::asio::read_at` пытается прочитать блок данных фиксированного размера, начиная со смещения;
- `boost::asio::read_until` пытается прочитать строку, пока не встретится разделитель, регулярное выражение или произвольный предикат.

Все три метода принимают сокет в качестве первого аргумента и объект буфера в качестве второго аргумента. Остальные аргументы являются необязательными и зависят от того, какая функция используется.

- *Условие завершения* — это функциональный объект, который принимает `error_code` и аргумент `size_t`. `error_code` будет установлен, если функция Asio обнаружила ошибку, а аргумент `size_t` соответствует количеству переданных байтов. Функциональный объект возвращает `size_t`, соответствующий количеству байтов, оставшихся для передачи, и возвращает 0, если операция завершена.
- *Условие соответствия* — это функциональный объект, который принимает диапазон, указанный начальным и конечным итератором. Он должен возвращать `std::pair`, где первый элемент является итератором, указывающим начальную точку для следующей попытки сопоставления, а второй элемент является логическим значением, представляющим, содержит ли диапазон совпадение.
- Ссылка `boost::system::error_code`, которая будет установлена функцией в случае возникновения ошибки.

В таблице 20.2 перечислены многие из способов, которыми можно вызвать одну из функций чтения.

Таблица 20.2. Аргументы для `read`, `read_at` и `read_until`

Вызов	Описание
`read(s, b, [cmp], [ec])`	Считывает определенное количество данных из сокета `s` в изменяемый буфер `b` в соответствии с условием завершения `cmp`. Устанавливает код ошибки `ec`, если возникла ошибка; в противном случае выдает системную ошибку
`ead_at(s, off, b, [cmp], [ec])`	Считывает определенный объем данных, начиная с сокета `s`, начиная со смещения `size_t off`, в изменяемый буфер `b` в соответствии с условием завершения `cmp`. Устанавливает код ошибки `ec`, если возникла ошибка; в противном случае выдает системную ошибку
`read_until(s, b, x, [ec])`	Считывает данные из сокета `s` в изменяемый буфер `b` до тех пор, пока он не выполнит условие, представленное `x`, которое может быть одним из следующих: `char`, `string_view`, `boost::regex` или условие соответствия. Устанавливает код ошибки `ec`, если возникла ошибка; в противном случае выдает системную ошибку

Также можно записывать данные в активный объект `socket` из буфера. Для блокировки записи Boost Asio предлагает две функции:

- `boost::asio::write` пытается записать блок данных фиксированного размера;
- `boost::asio::write_at` пытается записать блок данных фиксированного размера, начиная со смещения.

В таблице 20.3 показано, как вызвать эти методы. Их аргументы аналогичны аргументам для методов чтения.

Таблица 20.3. Аргументы для `write` и `write_at`

Вызов	Описание
`write(s, b, [cmp], [ec])`	Записывает определенное количество данных в сокет `s` из `const` буфера `b` в соответствии с условием завершения `cmp`. Устанавливает код ошибки `ec`, если возникла ошибка; в противном случае выдает системную ошибку
`write_at(s, off, b, [cmp], [ec])`	Записывает определенное количество данных из `const` буфера `b`, начиная с выключенного `size_t`, в сокет `s` в соответствии с условием завершения `cmp`. Устанавливает код ошибки `ec`, если возникла ошибка; в противном случае выдает системную ошибку

ПРИМЕЧАНИЕ

Существует множество перестановок для вызова функций чтения и записи. Перед включением Boost Asio в код внимательно изучите документацию.

Протокол передачи гипертекста (HTTP)

HTTP — это 30-летний протокол, лежащий в основе интернета. Хотя это очень сложный протокол для использования в Сети, его повсеместность делает его одним из наиболее подходящих вариантов. В следующем разделе мы будем использовать Boost Asio для выполнения очень простых HTTP-запросов. Необязательно иметь прочную основу для HTTP, поэтому можно пропустить этот раздел при первом чтении. Тем не менее информация здесь добавляет цвета к примерам в следующем разделе и предоставляет ссылки для дальнейшего изучения.

Сеансы HTTP имеют две стороны: клиент и сервер. HTTP-клиент отправляет незашифрованный запрос по TCP, содержащий одну или несколько строк, разделенных символом возврата каретки и переводом строки («новая строка CR-LF»).

Первая строка — это строка запроса, которая содержит три токена: метод HTTP, унифицированный указатель ресурса (URL) и версию запроса HTTP. Например, если клиент запрашивает файл с именем *index.htm*, строка состояния может быть: *GET /index.htm HTTP/1.1*.

Непосредственно после строки запроса находятся один или несколько *заголовков*, которые определяют параметры транзакции HTTP. Каждый заголовок содержит ключ и значение. Ключ должен состоять из буквенно-цифровых символов и тире. Двоеточие плюс пробел отделяет ключ от значения. Новая строка CR-LF завершает заголовок. Следующие заголовки особенно распространены в запросах:

- `Host` указывает домен запрашиваемой услуги. При желании можно включить порт. Например, `Host: www.google.com` указывает www.google.com в качестве хоста для запрашиваемой службы;
- `Accept` указывает допустимые типы носителей в формате MIME для ответа. Например, `Accept: text/plain` указывает, что отправитель запроса может обрабатывать открытый текст;
- `Accept-Language` определяет приемлемые человеческие языки для ответа. Например, `Accept-Language: en-US` указывает, что отправитель запроса может обрабатывать американский английский;
- `Accept-Encoding` указывает допустимые кодировки для ответа. Например, `Accept-Encoding: identity` указывает, что запрашивающая сторона может обрабатывать содержимое без какой-либо кодировки;
- `Connection` определяет параметры управления для текущего соединения. Например, `Connection: close` указывает, что соединение будет закрыто после завершения ответа.

Заголовки заканчиваются дополнительным символом CR-LF. Для определенных типов HTTP-запросов также нужно добавлять тело после заголовков. Если вы это сделаете, вы также включите заголовки `Content-Length` и `Content-Type`. Значение `Content-Length` указывает длину тела запроса в байтах, а значение `Content-Type` указывает формат MIME тела.

Первая строка ответа HTTP — это *строка состояния*, которая включает версию ответа HTTP, код состояния и соответствующее сообщение. Например, строка состояния `HTTP/1.1 200 OK` указывает на успешный («OK») запрос. Коды состояния всегда состоят из трех цифр. Первая цифра указывает на статусную группу кода:

- **1** (информационный)** — запрос получен;
- **2** (успешный)** — запрос получен и принят;
- **3** (перенаправление)** — требуются дальнейшие действия;
- **4** (ошибка клиента)** — запрос был неверным;
- **5** (ошибка сервера)** — запрос, кажется, в порядке, но сервер обнаружил внутреннюю ошибку.

После строки состояния ответ содержит любое количество заголовков в том же формате, что и ответ. Многие из одинаковых заголовков запросов также являются общими заголовками ответов. Например, если ответ HTTP содержит тело, заголовки ответа будут включать `Content-Length` и `Content-Type`.

Если нужно программировать HTTP-приложения, обязательно следует обратиться к библиотеке Boost Beast, которая предоставляет высокопроизводительные низкоуровневые средства HTTP и WebSockets. Она построена на вершине Asio и без проблем работает с ней.

ПРИМЕЧАНИЕ

Чтобы больше узнать о протоколе HTTP и его проблемах с безопасностью клиентов, почитайте книгу Михала Залевски (Michal Zalewski) *The Tangled Web: A Guide to Securing Modern Web Applications*. Все подробности см. в RFC 7230, 7231, 7232, 7233, 7234 и 7235 Инженерной группы интернета.

Реализация простого HTTP-клиента в Boost Asio

В этом разделе вы реализуете (очень) простой HTTP-клиент. Вы создаете HTTP-запрос, разрешите конечную точку, подключитесь к веб-серверу, напишете запрос и прочитаете ответ. Листинг 20.10 показывает одну возможную реализацию.

Листинг 20.10. Завершение простого запроса к веб-серверу киберкомандования армии США

```
#include <boost/asio.hpp>
#include <iostream>
#include <istream>
#include <ostream>
```

```
#include <string>

std::string request(std::string host, boost::asio::io_context& io_context) { ❶
  std::stringstream request_stream;
  request_stream << "GET / HTTP/1.1\r\n"
                    "Host: " << host << "\r\n"
                    "Accept: text/html\r\n"
                    "Accept-Language: en-us\r\n"
                    "Accept-Encoding: identity\r\n"
                    "Connection: close\r\n\r\n";
  const auto request = request_stream.str(); ❷
  boost::asio::ip::tcp::resolver resolver{ io_context };
  const auto endpoints = resolver.resolve(host, "http"); ❸
  boost::asio::ip::tcp::socket socket{ io_context };
  const auto connected_endpoint = boost::asio::connect(socket, endpoints); ❹
  boost::asio::write(socket, boost::asio::buffer(request)); ❺
  std::string response;
  boost::system::error_code ec;
  boost::asio::read(socket, boost::asio::dynamic_buffer(response), ec); ❻
  if (ec && ec.value() != 2) throw boost::system::system_error{ ec }; ❼
  return response;
}

int main() {
  boost::asio::io_context io_context;
  try {
    const auto response = request("www.arcyber.army.mil", io_context); ❽
    std::cout << response << "\n"; ❾
  } catch(boost::system::system_error& se) {
    std::cerr << "Error: " << se.what() << std::endl;
  }
}
```

```
--------------------------------------------------------------------------------
HTTP/1.1 200 OK
Pragma: no-cache
Content-Type: text/html; charset=utf-8
X-UA-Compatible: IE=edge
pw_value: 3ce3af822980b849665e8c5400e1b45b
Access-Control-Allow-Origin: *
X-Powered-By:
Server:
X-ASPNET-VERSION:
X-FRAME-OPTIONS: SAMEORIGIN
Content-Length: 76199
Cache-Control: private, no-cache
Expires: Mon, 22 Oct 2018 14:21:09 GMT
Date: Mon, 22 Oct 2018 14:21:09 GMT
Connection: close
<!DOCTYPE html>
<html lang="en-US">
<head id="Head">
--пропуск--
</body>
</html>
```

Сначала определяется функция запроса, которая принимает `host` и `io_context` и возвращает HTTP-ответ ❶. Во-первых, используется `std::stringstream` для построения `std::string`, содержащей HTTP-запрос ❷. Далее `host` разрешается с помощью `boost::asio::ip::tcp::resolver` ❸ и подключается `boost::asio::ip::tcp::socket` для получения диапазона конечной точки ❹. (Это соответствует подходу в листинге 20.4.)

Затем надо написать HTTP-запрос на сервер, к которому вы подключены. `boost::asio::write` используется, передавая подключенный сокет и запрос. Поскольку `write` принимает буферы Asio, `boost::asio::buffer` используется для создания `mutable_buffer` из запроса (который является `std::string`) ❺.

Далее вы читаете HTTP-ответ с сервера. Поскольку длина ответа заранее неизвестна, создается `std::string` с именем `response`, чтобы получить ответ. В конце концов, вы будете использовать это для резервного копирования динамического буфера. Для простоты HTTP-запрос содержит заголовок `Connection: close`, который заставляет сервер завершить соединение сразу после отправки своего ответа. Это приведет к тому, что Asio вернет код ошибки «конец файла» (значение 2). Поскольку такое поведение ожидаемо, объявляется `boost::system::error _code` для получения этой ошибки.

Затем вызывается `boost::asio::read` с подключенным сокетом, динамическим буфером, который получит ответ, и `error_condition` ❻.`boost::asio_dynamic_buffer` используется для создания динамического буфера из ответа. Сразу после возвращения `read` проверяется условие `error_condition`, отличное от конца файла (которое выбрасывается) ❼. В противном случае возвращается `response`.

В `main` вызывается функция запроса с хостом `www.arcyber.army.mil` и объектом `io_context` ❽. Наконец, ответ отправляется в стандартный вывод ❾.

Асинхронные чтение и запись

Также можно читать и записывать асинхронно с помощью Boost Asio. Соответствующие асинхронные функции аналогичны их блокирующим следствиям. Для асинхронного чтения Boost Asio предлагает три функции:

- `boost::asio::async_read` пытается прочитать блок данных фиксированного размера;
- `boost::asio::async_read_at` пытается прочитать блок данных фиксированного размера, начинающийся со смещения;
- `boost::asio::async_read_until` пытается читать до совпадения разделителя, регулярного выражения или произвольного предиката.

Boost Asio также предлагает две функции асинхронной записи:

- `boost::asio::async_write` пытается записать блок данных фиксированного размера;
- `boost::asio::async_write_at` пытается записать блок данных фиксированного размера, начинающийся со смещения.

Все пять из этих асинхронных функций принимают те же аргументы, что и их блокирующие аналоги, за исключением того, что их последний аргумент всегда является функциональным объектом обратного вызова, который принимает два аргумента: `boost::system::error_code`, указывающих, встретила ли функция ошибку, и `size_t`, указывающий количество переданных байтов. Для функций асинхронной записи необходимо определить, написал ли Asio всю полезную нагрузку. Поскольку эти вызовы являются асинхронными, поток не блокируется, пока ожидает завершения ввода-вывода. Вместо этого операционная система вызывает поток всякий раз, когда часть запроса ввода-вывода завершается.

Поскольку вторым аргументом обратного вызова является `size_t`, соответствующий количеству переданных байтов, можно выполнить арифметические действия, чтобы выяснить, есть ли у вас что-нибудь еще для записи. Если есть, нужно вызвать другую функцию асинхронной записи, передав оставшиеся данные.

Листинг 20.11 содержит асинхронную версию простого веб-клиента из листинга 20.10. Обратите внимание, что использование асинхронных функций немного сложнее. Но есть шаблон с обратными вызовами и обработчиками, который согласуется в течение всего срока действия запроса.

Листинг 20.11. Асинхронный рефакторинг листинга 20.9

```
#include <boost/asio.hpp>
#include <iostream>
#include <string>
#include <sstream>

using ResolveResult = boost::asio::ip::tcp::resolver::results_type;
using Endpoint = boost::asio::ip::tcp::endpoint;

struct Request {
  explicit Request(boost::asio::io_context& io_context, std::string host)
      : resolver{ io_context },
        socket{ io_context },
        host{ std::move(host) } { ❶
    std::stringstream request_stream;
    request_stream << "GET / HTTP/1.1\r\n"
                      "Host: " << this->host << "\r\n"
                      "Accept: text/plain\r\n"
                      "Accept-Language: en-us\r\n"
                      "Accept-Encoding: identity\r\n"
                      "Connection: close\r\n"
                      "User-Agent: C++ Crash Course Client\r\n\r\n";
    request = request_stream.str(); ❷
    resolver.async_resolve(this->host, "http",
       [this] (boost::system::error_code ec, const ResolveResult& results) {
         resolution_handler(ec, results); ❸
       });
  }

  void resolution_handler(boost::system::error_code ec,
```

```
                        const ResolveResult& results) {
  if (ec) { ❹
    std::cerr << "Error resolving " << host << ": " << ec << std::endl;
    return;
  }
  boost::asio::async_connect(socket, results,
          [this] (boost::system::error_code ec, const Endpoint& endpoint){
            connection_handler(ec, endpoint); ❺
          });
}

void connection_handler(boost::system::error_code ec,
                        const Endpoint& endpoint) { ❻
  if (ec) {
    std::cerr << "Error connecting to " << host << ": "
              << ec.message() << std::endl;
    return;
  }
  boost::asio::async_write(socket, boost::asio::buffer(request),
     [this] (boost::system::error_code ec, size_t transferred){
       write_handler(ec, transferred);
     });
}

void write_handler(boost::system::error_code ec, size_t transferred) { ❼
  if (ec) {
    std::cerr << "Error writing to " << host << ": " << ec.message()
              << std::endl;
  } else if (request.size() != transferred) {
    request.erase(0, transferred);
    boost::asio::async_write(socket, boost::asio::buffer(request),
                             [this] (boost::system::error_code ec,
                                     size_t transferred){
                               write_handler(ec, transferred);
                             });
  } else {
    boost::asio::async_read(socket, boost::asio::dynamic_buffer(response),
                            [this] (boost::system::error_code ec,
                                    size_t transferred){
                              read_handler(ec, transferred);
                            });
  }
}

void read_handler(boost::system::error_code ec, size_t transferred) { ❽
  if (ec && ec.value() != 2)
    std::cerr << "Error reading from " << host << ": "
              << ec.message() << std::endl;
}

const std::string& get_response() const noexcept {
  return response;
}
```

```
private:
  boost::asio::ip::tcp::resolver resolver;
  boost::asio::ip::tcp::socket socket;
  std::string request, response;
  const std::string host;
};

int main() {
  boost::asio::io_context io_context;
  Request request{ io_context, "www.arcyber.army.mil" };❾
  io_context.run(); ❿
  std::cout << request.get_response();
}
--------------------------------------------------------------------------------
HTTP/1.1 200 OK
Pragma: no-cache
Content-Type: text/html; charset=utf-8
X-UA-Compatible: IE=edge
pw_value: 3ce3af822980b849665e8c5400e1b45b
Access-Control-Allow-Origin: *
X-Powered-By:
Server:
X-ASPNET-VERSION:
X-FRAME-OPTIONS: SAMEORIGIN
Content-Length: 76199
Cache-Control: private, no-cache
Expires: Mon, 22 Oct 2018 14:21:09 GMT
Date: Mon, 22 Oct 2018 14:21:09 GMT
Connection: close

<!DOCTYPE html>
<html lang="en-US">
<head id="Head">
--пропуск--
</body>
</html>
```

Сначала объявляется класс `Request`, который будет обрабатывать веб-запрос. Он имеет единственный конструктор, который принимает `io_context`, и строку, содержащую хост, с которым нужно соединиться ❶. Как и в листинге 20.9, создается HTTP-запрос GET с использованием `std::stringstream` и результирующая строка сохраняется в поле запроса ❷. Далее используется `async_resolve` для запроса конечных точек, соответствующих запрошенному хосту. Внутри обратного вызова вызывается метод `resolution_handler` для текущего `Request` ❸.

`resolution_handler` получает обратный вызов от `async_resolve`. Сначала он проверяет состояние ошибки, выводя в стандартный вывод ошибки и возвращая результат, если будет найдена хотя бы одна ошибка ❹. Если `async_resolve` не передал ошибку, `resolution_handler` вызывает `async_connect`, используя конечные точки, содержащиеся в его переменной `results`. Он также передает поле сокета текущего запроса, в котором будет храниться соединение, которое `async_connect` собирается

создать. Наконец, он передает обратный вызов соединения в качестве третьего параметра. Внутри обратного вызова вызывается метод `connection_handler` текущего запроса ❺.

Обработчик `connection_handler` ❻ следует шаблону `resolution_handler`. Он проверяет наличие ошибки и, если она существует, выводит ее в стандартный вывод; в противном случае он обрабатывает запрос, вызывая `async_write`, который принимает три параметра: активный сокет, изменяемый оборачивающий буфер `request` и функцию обратного вызова. Функция обратного вызова, в свою очередь, вызывает метод `write_handler` для текущего запроса.

Видите шаблон в этих функциях обработчика? Оператор `write_handler` ❼ проверяет наличие ошибки и продолжает определять, был ли отправлен весь запрос. Если это не так, все равно нужно написать часть запроса, поэтому `request` соответствующим образом изменится и снова будет вызвана `async_write`. Если `async_write` записал весь запрос в `socket`, самое время прочитать ответ. Для этого вызывается `async_read`, используя `socket`, динамический буфер, оборачивающий поле `response`, и функцию обратного вызова, которая вызывает метод `read_handler` для текущего запроса.

`read_handler` ❽ сначала проверяет наличие ошибок. Поскольку в запросе использовался заголовок `Connection: close`, ожидается ошибка в конце файла (значение 2), как в листинге 20.10, которая в итоге игнорируется. Если возникает ошибка другого типа, она выводится в стандартный вывод ошибки и возвращается результат. На данный момент запрос завершен. (Уф!)

В `main` объявляется `io_context` и инициализируется запрос к `www.arcyber.army.mil` ❾. Поскольку используются асинхронные функции, вызывается метод `run` в `io_context` ❿. После возврата из `io_context` вы знаете, что асинхронные операции не ожидают, поэтому содержимое ответа объекта `Request` передается в стандартный вывод.

Создание сервера

Создание сервера поверх Boost Asio, по сути, аналогично созданию клиента. Чтобы принимать TCP-соединения, используется класс `boost::asio::ip::tcp::acceptor`, который принимает объект `boost::asio::io_context` в качестве единственного аргумента конструктора.

Чтобы принять TCP-соединение, используя блокирующий подход, применяется метод `accept` объекта `acceptor`; он принимает ссылку `boost::asio::ip::tcp::socket`, которая будет содержать сокет клиента, и необязательную ссылку `boost::error_code`, которая будет содержать любые возникшие ошибки. Если вы не предоставляете `boost::error_code` и возникает ошибка, `accept` выдаст вместо него `boost::system_error`. Как только вы примете возврат без ошибок, можно использовать переданный сокет для чтения и записи с теми же методами чтения и записи, которые использовались с клиентом в предыдущих разделах.

Например, в листинге 20.12 показано, как создать эхо-сервер, который получает сообщение и отправляет его обратно в верхнем регистре клиенту.

Листинг 20.12. Эхо-сервер в верхнем регистре

```
#include <iostream>
#include <string>
#include <boost/asio.hpp>
#include <boost/algorithm/string/case_conv.hpp>

using namespace boost::asio;

void handle(ip::tcp::socket& socket) { ❶
  boost::system::error_code ec;
  std::string message;
  do {
    boost::asio::read_until(socket, dynamic_buffer(message), "\n"); ❷
    boost::algorithm::to_upper(message); ❸
    boost::asio::write(socket, buffer(message), ec); ❹
    if (message == "\n") return; ❺
    message.clear();
  } while(!ec); ❻
}

int main() {
  try {
    io_context io_context;
    ip::tcp::acceptor acceptor{ io_context,
                                ip::tcp::endpoint(ip::tcp::v4(), 1895) }; ❼
    while (true) {
      ip::tcp::socket socket{ io_context };
      acceptor.accept(socket); ❽
      handle(socket); ❾
    }
  } catch (std::exception& e) {
    std::cerr << e.what() << std::endl;
  }
}
```

Объявляется функция `handle`, которая принимает ссылку на `socket`, соответствующую клиенту, и обрабатываются сообщения от нее ❶. В цикле `do-while` читается строка текста из клиента в строку с именем `message` ❷, она конвертируется в верхний регистр с помощью функции `to_upper`, показанной в листинге 15.31 ❸, и записывается обратно клиенту ❹. Если клиент отправил пустую строку, дескриптор завершается ❺; в противном случае очищается содержимое сообщения и цикла, если не возникло условия ошибки ❻.

Внутри `main` инициализируется `io_context` и `acceptor`, чтобы программа связывалась с сокетом `localhost: 1895` ❼. Внутри бесконечного цикла создается сокет и вызывается `accept` в `acceptor` ❽. Пока это не вызывает исключения, сокет будет

представлять нового клиента, и можно будет передать этот сокет в `handle` для обслуживания запроса ❾.

ПРИМЕЧАНИЕ

В листинге 20.12 было выбрано прослушивание порта 1895. Этот выбор технически несущественен, поскольку никакая другая программа, работающая на компьютере, в настоящее время не использует этот порт. Тем не менее существуют рекомендации о том, как решить, какой порт будет прослушивать программа. IANA публикует список зарегистрированных портов по адресу www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.txt, которых лучше избегать. Кроме того, современные операционные системы обычно требуют, чтобы программа имела повышенные привилегии для привязки к порту со значением 1023 или ниже (системный порт). Порты с 1024 по 49151 обычно не требуют повышенных привилегий и называются пользовательскими портами. Порты с 49152 по 65535 являются динамическими/частными портами, которые в целом безопасны для использования, поскольку они не будут зарегистрированы в IANA.

Для взаимодействия с сервером в листинге 20.12 можно использовать сетевую утилиту *GNU Netcat*, которая позволяет создавать входящие и исходящие TCP иUDP-соединения, а затем читать и записывать данные. Если вы используете Unix-подобную систему, возможно, она уже установлена. Если нет, см. nmap.org/ncat/.В листинге 20.13 показан пример сеанса, который подключается к эхо-серверу верхнего регистра.

Листинг 20.13. Взаимодействие с эхо-сервером верхнего регистра с использованием Netcat

```
$ ncat localhost 1895 ❶
The 300 ❷
THE 300
This is Blasphemy! ❷
THIS IS BLASPHEMY!
This is madness! ❷
THIS IS MADNESS!
Madness...? ❷
MADNESS...?
This is Sparta! ❷
THIS IS SPARTA!
❸
Ncat: Broken pipe. ❹
```

Netcat (`ncat`) принимает два аргумента: хост и порт ❶. После вызова программы каждая введенная строка приводит к получению с сервера заглавных букв. После ввода текста в стандартный ввод Netcat отправляет его на сервер ❷, который отвечает заглавными буквами. Как только вы отправите ему пустую строку ❸, сервер завершит работу сокета, и вы получите сообщение `Broken pipe` ❹.

Чтобы принимать соединения с использованием асинхронного подхода, используется метод `async_accept` на `acceptor`. Он принимает один аргумент: объект обратного вызова, который принимает `error_code` и `socket`. Если возникает ошибка,

`error_code` содержит условие ошибки; в противном случае `socket` представляет собой успешно подключенного клиента. Поэтому можно использовать `socket` так же, как и при блокировке.

Обычным шаблоном для асинхронных серверов, ориентированных на установление соединения, является использование шаблона `std::enable_shared_from_this`, описанного в разделе «Продвинутые шаблоны» на с. 436. Идея состоит в том, чтобы создать общий указатель на объект сеанса для каждого соединения. При регистрации обратных вызовов для чтения и записи в объекте сеанса захватывается общий указатель «из `this`» внутри объекта обратного вызова, так что, пока ввод/вывод находится в режиме ожидания, сеанс остается живым. Когда ввод-вывод не ожидается, объект сеанса уничтожается вместе со всеми общими указателями. В листинге 20.14 показано, как переопределить эхо-сервер в верхнем регистре, используя асинхронный ввод-вывод.

Листинг 20.14. Асинхронная версия листинга 20.12

```
#include <iostream>
#include <string>
#include <boost/asio.hpp>
#include <boost/algorithm/string/case_conv.hpp>
#include <memory>

using namespace boost::asio;

struct Session : std::enable_shared_from_this<Session> {
  explicit Session(ip::tcp::socket socket) : socket{ std::move(socket) } { } ❶
  void read() {
    async_read_until(socket, dynamic_buffer(message), '\n', ❷
            [self=shared_from_this()] (boost::system::error_code ec,
                                       std::size_t length) {
              if (ec || self->message == "\n") return; ❸
              boost::algorithm::to_upper(self->message);
              self->write();
            });
  }
  void write() {
    async_write(socket, buffer(message), ❹
                [self=shared_from_this()] (boost::system::error_code ec,
                                           std::size_t length) {
                  if (ec) return; ❺
                  self->message.clear();
                  self->read();
                });
  }
private:
  ip::tcp::socket socket;
  std::string message;
}

void serve(ip::tcp::acceptor& acceptor) {
```

```
  acceptor.async_accept([&acceptor](boost::system::error_code ec, ❻
                                    ip::tcp::socket socket) {
    serve(acceptor); ❼
    if (ec) return;
    auto session = std::make_shared<Session>(std::move(socket)); ❽
    session->read();
  });
}

int main() {
  try {
    io_context io_context;
    ip::tcp::acceptor acceptor{ io_context,
                                ip::tcp::endpoint(ip::tcp::v4(), 1895) };
    serve(acceptor);
    io_context.run(); ❾
  } catch (std::exception& e) {
    std::cerr << e.what() << std::endl;
  }
}
```

Сначала определяется класс `Session` для управления соединениями. Внутри конструктора принимается во владение `socket`, соответствующий подключающемуся клиенту, и сохраняется как член ❶.

Затем объявляется метод `read`, который вызывает `async_read_until` в `socket`, чтобы тот считывал в `dynamic_buffer`, упаковывая `string message` член до следующего символа новой строки `\n` ❷. Объект обратного вызова захватывает его как `shared_ptr` с помощью метода `shared_from_this`. При вызове функция проверяет либо условие ошибки, либо пустую строку, и в этом случае возвращается результат ❸. В противном случае обратный вызов преобразует сообщение в верхний регистр и вызывает метод `write`.

Метод `write` аналогичен методу `read`. Он вызывает `async_read`, передавая `socket`, `message` (теперь заглавными буквами) и функцию обратного вызова ❹. Внутри функции обратного вызова проверяется наличие ошибки и немедленно возвращается результат, если она существует ❺. В противном случае вы знаете, что Asio успешно отправил `message` в верхнем регистре клиенту, поэтому вызывается команда `clear` для него, чтобы подготовить следующее сообщение от клиента. Затем вызывается метод `read`, который запускает процесс заново.

Затем определяется функция `serve`, которая принимает объект `acceptor`. Внутри функции вызывается `async_accept` для объекта `acceptor` и передается функция обратного вызова для обработки соединений ❻. Функция обратного вызова сначала снова вызывает `serve`, поэтому программа может обрабатывать новые соединения немедленно ❼. Это секретный ингредиент, который делает асинхронную обработку настолько мощной со стороны сервера: можно обрабатывать много соединений одновременно, потому что выполняющемуся потоку не нужно обслуживать одного клиента перед обработкой другого. Затем проверяется наличие ошибки, и работа

завершается, если ошибка найдена; в противном случае создается `shared_ptr`, которому принадлежит новый объект `Session` ❽. Этот объект `Session` будет иметь `socket`, который `acceptor` только что установил. Вызывается метод `read` для нового объекта `Session`, который создает вторую ссылку в `shared_ptr` благодаря захвату `shared_from_this`. Теперь все готово! Как только цикл `read` и `write` заканчивается из-за пустой строки от клиента или некоторого состояния ошибки, ссылка `shared_ptr` обнуляется, а объект `Session` уничтожается.

Наконец, в `main` создается `io_context` и `acceptor`, как в листинге 20.12. Затем `acceptor` передается в функцию `serve`, чтобы начать цикл обслуживания, и вызывается `run` для `io_context`, чтобы начать обработку асинхронных операций ❾.

Конкурентный режим в Boost Asio

Чтобы сделать программу Boost Asio конкурентной, можно просто порождать задачи, которые вызывают запуск объекта `io_context`. Конечно, это не делает программу безопасной, и все указания в разделе «Совместное использование и координирование» на с. 744 действуют в полном объеме. В листинге 20.15 показано, как настроить конкурентность сервера из листинга 20.14.

Листинг 20.15. Конкурентность асинхронного эхо-сервера

```
#include <iostream>
#include <string>
#include <boost/asio.hpp>
#include <boost/algorithm/string/case_conv.hpp>
#include <memory>
#include <future>

struct Session : std::enable_shared_from_this<Session> {
--пропуск--
};

void serve(ip::tcp::acceptor& acceptor) {
--пропуск--
}

int main() {
  const int n_threads{ 4 };
  boost::asio::io_context io_context{ n_threads };
  ip::tcp::acceptor acceptor{ io_context,
                              ip::tcp::endpoint(ip::tcp::v4(), 1895) }; ❶
  serve(acceptor); ❷

  std::vector<std::future<void>> futures;
  std::generate_n(std::back_inserter(futures), n_threads, ❸
                  [&io_context] {
                    return std::async(std::launch::async,
                                      [&io_context] { io_context.run(); }); ❹
                  });
```

```
    for(auto& future : futures) { ❺
      try {
        future.get(); ❻
      } catch (const std::exception& e) {
        std::cerr << e.what() << std::endl;
      }
    }
  }
```

Определения `Session` и `serve` идентичны. Внутри `main` объявляется постоянная `n_threads`, представляющая число потоков, которые будут использованы для обслуживания, `io_context` и `acceptor` с параметрами, идентичными тем, которые перечислены в листинге 12.12 ❶. Затем вызывается `serve` для запуска цикла `async_accept` ❷.

Более или менее `main` почти идентичен листингу 12.12. Разница в том, что будут использоваться несколько потоков для запуска `io_context`, а не один. Сначала инициализируется вектор для хранения каждого `future`, соответствующего запускаемым задачам. Во-вторых, используется аналогичный подход с `std::generate_n` для создания задач ❸. В качестве объекта порождающей функции передается лямбда-выражение, которое вызывает `std::async` ❹. В вызове `std::async` передается политика выполнения `std::launch::async`, и вызываемый функциональный объект запускается в `io_context`.

Boost Asio теперь не находится в состоянии гонки, потому что выполнено несколько задач для запуска `io_context`. Нужно дождаться завершения всех асинхронных операций, поэтому вызывается `get` для каждого `future`, которое хранится в `futures` ❺. После завершения этого цикла каждый `Request` завершается, и сводка полученных ответов становится готовой к выводу ❻.

Иногда имеет смысл создать дополнительные потоки и назначить их для обработки ввода-вывода. Часто одного потока будет достаточно. Нужно измерить, оправданна ли оптимизация (и сопутствующие трудности, возникающие из-за конкурентного кода).

Итоги

В этой главе описана Boost Asio, библиотека для низкоуровневого программирования ввода/вывода. Вы изучили основы постановки в очередь асинхронных задач и предоставления пула потоков в Asio, а также способы взаимодействия с основными сетевыми средствами. Вы создали несколько программ, в том числе простой HTTP-клиент, с использованием синхронного и асинхронного подходов и эхо-сервер.

Упражнения

20.1. Используйте документацию Boost Asio для изучения аналогов классов UDP и классов TCP, о которых вы узнали в этой главе. Перепишите эхо-сервер перевода в верхний регистр в листинге 20.14 как службу UDP.

20.2. Используйте документацию Boost Asio для изучения классов ICMP. Напишите программу, которая получает информацию из всех хостов в данной подсети, чтобы выполнить анализ сети. Изучите Nmap, программу для картирования сети, доступную бесплатно на **nmap.org**.

20.3. Изучите документацию Boost Beast. Перепишите листинги 20.10 и 20.11, используя Beast.

20.4. Используйте Boost Beast, чтобы написать HTTP-сервер, который обслуживает файлы из каталога. Для получения справки см. примеры проектов Boost Beast, доступные в документации.

Что еще почитать?

- «The TCP/IP Guide», Charles M. Kozierok (No Starch Press, 2005)
- «Tangled Web: A Guide to Securing Modern Web Applications», Michal Zalewski (No Starch Press, 2012)
- «The Boost C++ Libraries», 2nd Edition, Boris Schäling (XML Press, 2014)
- «Boost.Asio C++ Network Programming», Wisnu Anggoro, John Torjo (Packt, 2015)

21 Создание приложений

Для группы безволосых обезьян нам действительно удалось изобрести несколько довольно невероятных вещей.

Эрнест Клайн, «Первому игроку приготовиться»

Эта глава содержит ряд важных тем, которые добавят реальных знаний о C++ и научат основам создания настоящих приложений. Она начинается с обсуждения поддержки программы, встроенной в C++, которая позволяет взаимодействовать с жизненным циклом приложения. Далее вы узнаете о Boost Program Options, превосходной библиотеке для разработки консольных приложений. Она содержит средства для приема ввода от пользователей без необходимости изобретать велосипед. Кроме того, вы узнаете некоторые особенности препроцессора и компилятора, с которыми, вероятно, столкнетесь при создании приложения, исходный код которого превышает один файл.

Поддержка программ

Иногда программы должны взаимодействовать с жизненным циклом приложения вашей ОС. В этом разделе рассматриваются три основные категории таких взаимодействий:

- обработка завершения программы и ее очистка;
- связь с окружающей средой;
- управление сигналами операционной системы.

Чтобы показать различные возможности из этого раздела, в качестве основы будем использовать листинг 21.1. Там используется расширенный аналог класса `Tracer` из

листинга 4.5, чтобы отследить, какие объекты очищаются в различных сценариях завершения программы.

Листинг 21.1. Основа для исследования возможностей завершения и очистки программы

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

struct Tracer { ❶
  Tracer(std::string name_in)
    : name{ std::move(name_in) } {
    std::cout << name << " constructed.\n";
  }
  ~Tracer() {
    std::cout << name << " destructed.\n";
  }
private:
  const std::string name;
};

Tracer static_tracer{ "static Tracer" }; ❷

void run() { ❸
  std::cout << "Entering run()\n";
  // ...
  std::cout << "Exiting run()\n";
}

int main() {
  std::cout << "Entering main()\n"; ❹
  Tracer local_tracer{ "local Tracer" }; ❺
  thread_local Tracer thread_local_tracer{ "thread_local Tracer" }; ❻
  const auto* dynamic_tracer = new Tracer{ "dynamic Tracer" }; ❼
  run(); ❽
  delete dynamic_tracer; ❾
  std::cout << "Exiting main()\n"; ❿
}
--------------------------------------------------------------------------------
static Tracer constructed. ❷
Entering main() ❹
local Tracer constructed. ❺
thread_local Tracer constructed. ❻
dynamic Tracer constructed.❼
Entering run() ❽
Exiting run() ❽
dynamic Tracer destructed. ❾
Exiting main() ❿
local Tracer destructed. ❺
thread_local Tracer destructed. ❻
static Tracer destructed. ❷
```

Сначала объявляется класс `Tracer`, который принимает произвольный тег `std::string` и сообщает стандартному выводу, когда объект `Tracer` создается и уничтожается ❶. Затем объявляется `Tracer` со статической длительностью хранения ❷. Функция `run` сообщает, когда программа вошла и вышла из нее ❸. В середине находится один комментарий, который вы замените другим кодом в следующих разделах. В `main` происходит объявление ❹; объекты `Tracer` инициализируются локальной ❺, потоковой ❻ и динамической ❼ длительностью хранения и вызывается `run` ❽. Затем динамический объект `Tracer` удаляется ❾ и объявляется, что `main` собирается завершить работу ❿.

ПРЕДУПРЕЖДЕНИЕ

Если какой-либо вывод из листинга 21.1 кажется странным, просмотрите раздел «Длительность хранения объекта» на с. 148, прежде чем продолжать!

Обработка завершения программы и очистка

Заголовок `<cstdlib>` содержит несколько функций для управления завершением программы и очисткой ресурса. Существует две широкие категории функций завершения программы:

- те, которые вызывают завершение программы;
- те, которые регистрируют обратный вызов, когда должно произойти завершение.

Завершение обратного вызова с помощью std::atexit

Чтобы зарегистрировать функцию, вызываемую при нормальном завершении программы, используется функция `std::atexit`. Можно зарегистрировать несколько функций, и они будут вызываться в обратном порядке после их регистрации. Функции обратного вызова не принимают аргументов и возвращают `void`. Если `std::atexit` успешно зарегистрирует функцию, она вернет ненулевое значение; в противном случае возвращается ноль.

В листинге 21.2 показано, что можно зарегистрировать обратный вызов `atexit`, и он будет вызван в ожидаемый момент.

Листинг 21.2. Регистрация обратного вызова atexit

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

struct Tracer {
--пропуск--
};

Tracer static_tracer{ "static Tracer" };
void run() {
```

```
  std::cout << "Registering a callback\n"; ❶
  std::atexit([] { std::cout << "***std::atexit callback executing***\n"; }); ❷
  std::cout << "Callback registered\n"; ❸
}

int main() {
--пропуск--
}
```

```
--------------------------------------------------------------------------------
static Tracer constructed.
Entering main()
local Tracer constructed.
thread_local Tracer constructed.
dynamic Tracer constructed.
Registering a callback
Callback registered ❸
dynamic Tracer destructed.
Exiting main()
local Tracer destructed.
thread_local Tracer destructed.
***std::atexit callback executing*** ❷
static Tracer destructed.
```

В `run` вы объявляете, что собираетесь зарегистрировать обратный вызов ❶, делаете это ❷, а затем объявляете, что собираетесь вернуться из `run` ❸. В выводе ясно видно, что обратный вызов происходит после того, как `main` завершает работу и все нестатические объекты уничтожаются.

При программировании функции обратного вызова нужно учесть два важных предупреждения:

- не стоит выбрасывать неперехваченное исключение из функции обратного вызова. Это приведет к вызову `std::terminate`;
- будьте осторожны при взаимодействии с нестатическими объектами в программе. Функции обратного вызова `atexit` выполняются после основных возвратов, поэтому все локальные, потоковые и динамические объекты будут уничтожены в этот момент, если вы не позаботитесь о том, чтобы сохранить их живыми.

ПРИМЕЧАНИЕ

С помощью `std::atexit` можно зарегистрировать как минимум 32 функции, хотя точное ограничение определяется реализацией.

Выход с помощью std::exit

На протяжении всей книги мы прерывали программы, возвращая результат из `main`. В некоторых случаях, например в конкурентных программах, может потребоваться корректный выход из программы каким-либо другим способом, хотя следует избегать возникновения связанных с этим сложностей. Можно использовать функцию

`std::exit`, которая принимает один `int`, соответствующий коду завершения программы. Он выполнит следующие шаги очистки:

1. Локальные объекты потока, связанные с текущим потоком, и статические объекты будут уничтожены. Запускаются любые функции обратного вызова `atexit`.
2. Все стандартные ввод, вывод и вывод ошибок сбрасываются.
3. Любые временные файлы удаляются.
4. Программа сообщает заданный код состояния операционной среде, которая возобновляет управление.

Листинг 21.3 показывает поведение `std::exit`, регистрируя обратный вызов `atexit` и вызывая выход изнутри `run`.

Листинг 21.3. Вызов std::exit

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

struct Tracer {
--пропуск--
};

Tracer static_tracer{ "static Tracer" };

void run() {
  std::cout << "Registering a callback\n"; ❶
  std::atexit([] { std::cout << "***std::atexit callback executing***\n"; }); ❷
  std::cout << "Callback registered\n"; ❸
  std::exit(0); ❹
}

int main() {
--пропуск--
}
--------------------------------------------------------------------------------
static Tracer constructed.
Entering main()
local Tracer constructed.
thread_local Tracer constructed.
dynamic Tracer constructed.
Registering a callback ❶
Callback registered ❸
thread_local Tracer destructed.
***std::atexit callback executing*** ❹
static Tracer destructed.
```

В ходе выполнения вы объявляете, что регистрируете обратный вызов ❶, регистрируете его с помощью `atexit` ❷, затем объявляете, что регистрация завершена ❸,

и вызываете `exit` с нулевым аргументом ❹. Сравните вывод программы из листинга 21.3 с выводом из листинга 21.2. Обратите внимание, что следующие строки не отображаются:

```
dynamic Tracer destructed.
Exiting main()
local Tracer destructed.
```

В соответствии с правилами для `std::exit` локальные переменные в стеке вызовов не очищаются. И, конечно же, поскольку программа никогда не возвращается в главное состояние после запуска, `delete` никогда не вызывается.

Этот пример подчеркивает важное соображение: не стоит использовать `std::exit` для обработки нормального выполнения программы. Это упомянуто здесь для полноты, потому что такое поведение можно было заметить в более раннем коде C++.

ПРИМЕЧАНИЕ

Заголовок `<cstdlib>` также включает в себя `std::quick_exit`, запускающий обратные вызовы, которые регистрируются с помощью `std::at_quick_exit` с интерфейсом, аналогичным `std::atexit`. Основное отличие состоит в том, что обратные вызовы `at_quick_exit` не будут выполняться, если явно не вызвать `quick_exit`, тогда как обратные вызовы `atexit` всегда будут выполняться, когда программа собирается завершить работу.

std::abort

Чтобы завершить программу, можно использовать ядерную опцию — `std::abort`. Эта функция принимает один целочисленный код состояния и немедленно возвращает его в операционную среду. Никакие объектные деструкторы не вызываются и никакие обратные вызовы `std::atexit` не запускаются. В листинге 21.4 показано, как использовать `std::abort`.

Листинг 21.4. Вызов std::abort

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

struct Tracer {
--пропуск--
};

Tracer static_tracer{ "static Tracer" };

void run() {
  std::cout << "Registering a callback\n"; ❶
  std::atexit([] { std::cout << "***std::atexit callback executing***\n"; }); ❷
  std::cout << "Callback registered\n"; ❸
  std::abort(); ❹
}
```

```
int main() {
--пропуск--
}
-------------------------------------------------------------------------------
static Tracer constructed.
Entering main()
local Tracer constructed.
thread_local Tracer constructed.
dynamic Tracer constructed.
Registering a callback
Callback registered
```

В `run` снова объявляется, что регистрируется обратный вызов ❶ с помощью `atexit` ❷ и объявляется, что регистрация завершена ❸. На этот раз вызывается `abort` ❹. Обратите внимание, что после того как вы объявили, что завершили регистрацию обратного вызова ❶, ничего не выводится. Программа не очищает какие-либо объекты, и обратный вызов `atexit` не запускается.

Как вы можете себе представить, для `std::abort` существует не так уж много канонического использования. Основное, с чем можно столкнуться, — это стандартное поведение `std::terminate`, которое вызывается, когда два исключения находятся в полете одновременно.

Коммуникация с окружающей средой

Иногда может понадобиться создать другой процесс. Например, браузер Google Chrome запускает множество процессов для обслуживания одного сеанса браузера.

Это обеспечивает некоторую безопасность и надежность благодаря совмещению модели процессов операционной системы. Например, веб-приложения и плагины запускаются в отдельных процессах, поэтому при сбое весь браузер не падает. Кроме того, запуская механизм рендеринга браузера в отдельном процессе, любые уязвимости безопасности становится все труднее использовать, поскольку Google блокирует разрешения этого процесса в так называемой изолированной среде.

std::system

Можно запустить отдельный процесс с помощью функции `std::system` в заголовке `<cstdlib>`, которая принимает строку в стиле C, соответствующую команде, которую нужно выполнить, и возвращает `int`, соответствующий коду возврата из команды. Фактическое поведение зависит от операционной среды. Например, функция будет вызывать cmd.exe на компьютере с Windows и */bin/sh* на компьютере с Linux. Эта функция блокируется, пока команда еще выполняется.

В листинге 21.5 показано, как использовать `std::system` для проверки связи с удаленным хостом. (Необходимо обновить содержимое команды до соответствующей команды для вашей операционной системы, если не используется Unix-подобная ОС.)

Сначала инициализируется строка `command`, содержащая `ping -c 4 google.com` ❶. Затем вызывается `std::system`, передавая содержимое `command` ❷. Это заставляет операционную систему вызывать команду `ping` с аргументом `-c 4`, который задает четыре запрашивания информации, и адрес `google.com`. Затем выводится сообщение о состоянии, говорящее о возвращаемом значении из `std::system` ❸.

Листинг 21.5. Использование std::system для вызова утилиты ping (вывод из macOS Mojave версии 10.14.)

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

int main() {
  std::string command{ "ping -c 4 google.com" }; ❶
  const auto result = std::system(command.c_str()); ❷
  std::cout << "The command \'" << command
            << "\' returned " << result << "\n";
}
-------------------------------------------------------------------------------
PING google.com (172.217.15.78): 56 data bytes
64 bytes from 172.217.15.78: icmp_seq=0 ttl=56 time=4.447 ms
64 bytes from 172.217.15.78: icmp_seq=1 ttl=56 time=12.162 ms
64 bytes from 172.217.15.78: icmp_seq=2 ttl=56 time=8.376 ms
64 bytes from 172.217.15.78: icmp_seq=3 ttl=56 time=10.813 ms
--- google.com ping statistics ---
4 packets transmitted, 4 packets received, 0.0% packet loss
round-trip min/avg/max/stddev = 4.447/8.950/12.162/2.932 ms
The command 'ping -c 4 google.com' returned 0 ❸
```

std::getenv

Операционные среды обычно имеют *переменные среды*, которые пользователи и разработчики могут установить, чтобы помочь программам находить важную информацию, необходимую для запуска программ. Заголовок `<cstdlib>` содержит функцию `std::getenv`, которая принимает строку в стиле C, соответствующую имени переменной среды, которую нужно найти, и возвращает строку в стиле C с содержимым соответствующей переменной. Если такая переменная не найдена, функция возвращает `nullptr`.

В листинге 21.6 показано, как использовать `std::getenv` для получения переменной пути, содержащей список каталогов, в которых находятся важные исполняемые файлы.

Листинг 21.6. Использование std::getenv для получения переменной пути. (Вывод из macOS Mojave версии 10.14.)

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

int main() {
```

```
  std::string variable_name{ "PATH" }; ❶
  std::string result{ std::getenv(variable_name.c_str()) }; ❷
  std::cout << "The variable " << variable_name
            << " equals " << result << "\n"; ❸
}
---------------------------------------------------------------------------
The variable PATH equals /usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin
```

Сначала инициализируется строка `variable_name`, содержащая `PATH` ❶. Затем результат вызова `std::getenv` с `PATH` сохраняется в строку с именем `result` ❷. Затем выводятся результаты в стандартный вывод ❸.

Управление сигналами операционной системы

Сигналы ОС — это асинхронные уведомления, отправляемые процессам, которые оповещают программу о том, что произошло событие. Заголовок `<csignal>` содержит шесть макропостоянных, которые представляют разные сигналы от ОС к программе (эти сигналы не зависят от ОС).

- `SIGTERM` представляет запрос завершения.
- `SIGSEGV` представляет неверный доступ к памяти.
- `SIGINT` представляет внешнее прерывание, такое как прерывание клавиатуры.
- `SIGILL` представляет недопустимый образ программы.
- `SIGABRT` представляет ненормальное условие завершения, такое как `std::abort`.
- `SIGFPE` представляет ошибку числа с плавающей точкой, например деление на ноль.

Чтобы зарегистрировать обработчик для одного из этих сигналов, используется функция `std::signal` в заголовке `<csignal>`. В качестве первого аргумента она принимает значение `int`, соответствующее одному из перечисленных сигнальных макросов. Вторым аргументом является функциональный указатель (а не функциональный объект!) на функцию, которая принимает `int`, соответствующий макросу сигнала и возвращающий `void`. Эта функция должна иметь привязку к C (хотя большинство реализаций также разрешает привязку к C++). Вы узнаете о связи C позже в этой главе. Пока просто добавьте `extern "C"` к определению функции. Обратите внимание, что из-за асинхронного характера прерываний любой доступ к глобальному изменяемому состоянию должен быть синхронизирован.

В листинге 21.7 приведена программа, ожидающая прерывания клавиатуры.

Листинг 21.7. Регистрация прерываний с клавиатуры с помощью std::signal

```
#include <csignal>
#include <iostream>
#include <chrono>
#include <thread>
#include <atomic>
```

```
std::atomic_bool interrupted{}; ❶

extern "C" void handler(int signal) {
  std::cout << "Handler invoked with signal " << signal << ".\n"; ❷
  interrupted = true; ❸
}

int main() {
  using namespace std::chrono_literals;
  std::signal(SIGINT, handler); ❹
  while(!interrupted) { ❺
    std::cout << "Waiting..." << std::endl; ❻
    std::this_thread::sleep_for(1s);
  }
  std::cout << "Interrupted!\n"; ❼
}
-------------------------------------------------------------------------------
Waiting...
Waiting...
Waiting...
Handler invoked with signal 2.
Interrupted! ❼
```

Сначала объявляется переменная `atomic_bool` с именем `interrupted`, в которой хранится информация о том, получила ли программа прерывание с клавиатуры ❶ (она имеет статическую длительность хранения, потому что нельзя использовать функциональные объекты в `std::signal` и, следовательно, нужно использовать функцию, не являющуюся членом, для обработки обратного вызова). Затем объявляется обработчик обратного вызова, который принимает вызываемый `int` сигнал, печатает его значение в стандартный вывод ❷ и устанавливает прерывание равным `true` ❸.

В `main` устанавливается обработчик сигнала для кода прерывания `SIGINT` в `handler` ❹. Внутри цикла ожидается прерывания программы ❺ с выводом сообщения ❻ и секундным «засыпанием» ❼. Как только программа была прервана, выводится сообщение и завешается работа `main` ❼.

ПРИМЕЧАНИЕ

Обычно прерывание в современных операционных системах можно вызвать с клавиатуры, нажав `Ctrl+C`.

ProgramOptions в Boost

Большинство консольных приложений принимают параметры командной строки. Как вы узнали из раздела «Три перегрузки main» на с. 342, можно определить `main`, чтобы принимать параметры `argc` и `argv`, которые операционная среда будет заполнять количеством аргументов и их содержимым соответственно. Всегда можно

проанализировать их вручную и соответствующим образом изменить вашу программу, но есть и лучший способ: библиотека Boost ProgramOptions является важным компонентом для написания консольных приложений.

ПРИМЕЧАНИЕ

Все классы Boost ProgramOptions, представленные в этом разделе, доступны в заголовке `<boost/program_options.hpp>`.

У вас может возникнуть желание написать собственный код для анализа аргументов, но ProgramOptions — более разумный выбор по четырем причинам:

1. **Это гораздо удобнее.** Изучив краткий декларативный синтаксис ProgramOptions, можно легко описать довольно сложные интерфейсы консоли в нескольких строках кода.
2. **Она легко обрабатывает ошибки.** Когда пользователь злоупотребляет вашей программой, ProgramOptions сообщает пользователю, как он неправильно использовал программу без каких-либо дополнительных усилий с вашей стороны.
3. **Она автоматически генерирует подсказку.** На основе декларативной разметки ProgramOptions создает красиво отформатированную, простую в использовании документацию от вашего имени.
4. **Она выходит за пределы командной строки.** Если нужно создать конфигурацию из конфигурационных файлов или переменных среды, это легко сделать, отойдя от аргументов командной строки.

ProgramOptions состоит из трех частей:

1. **Описание параметров** позволяет указать разрешенные параметры.
2. **Компонент анализаторов** извлекает имена и значения параметров из командной строки, файлов конфигурации и переменных среды.
3. **Компонент хранения** предоставляет интерфейс для доступа к типизированным опциям.

В последующих подразделах вы узнаете о каждой этой части.

Описание опций

Три основных класса составляют компонент описания параметров:

- `boost::program_options::option_description` описывает одну опцию;
- `boost::program_options::value_semantic` знает желаемый тип отдельной опции;
- `boost::program_options::options_description` — это контейнер для нескольких объектов типа `option_description`.

`options_description` создается, чтобы указать описание для параметров программы. При желании можно добавить в конструктор один строковый аргумент, который описывает вашу программу. Он выведется в описании, если его добавить, но это не будет иметь никакого функционального влияния. Затем используется его метод `add_options`, который возвращает специальный тип объекта типа `boost::program_options::options_description_easy_init`. Этот класс имеет специальный `operator()`, который принимает как минимум два аргумента.

Первый аргумент — это название опции, которую нужно добавить. ProgramOptions очень умная, поэтому можно указать длинное имя и короткое имя, разделенные запятой. Например, если есть опция с именем `threads`, ProgramOptions привязывает параметр `--threads` из командной строки к этой опции. Если вместо этого вы назвали опцию `threads,t`, ProgramOptions привяжет либо `--threads`, либо `-t` к вашему варианту. Второй аргумент — это описание опции. Можно использовать `value_semantic`, описание стоки в стиле C или и то и другое. Поскольку `options_description_easy_init` возвращает ссылку на себя из `operator()`, можно объединить эти вызовы, чтобы сформировать сжатое представление параметров вашей программы. Как правило, объекты `value_semantic` не создаются напрямую.

Вместо этого используется удобная шаблонная функция `boost::program_options::value` для их генерации. Она принимает один параметр шаблона, соответствующий желаемому типу опции. Полученный указатель указывает на объект, который имеет код для приведения ввода текста (например, из командной строки) в нужный тип. Например, чтобы указать параметр типа `int`, нужно вызвать значение `<int>()`.

Получившийся объект, на который направлен указатель, будет иметь несколько методов, которые позволят указать дополнительную информацию о параметре. Например, можно использовать метод `default_value`, чтобы установить значение параметра по умолчанию. Чтобы указать, что для параметра типа `int` по умолчанию должно быть задано значение 42, используйте следующую конструкцию:

```
value<int>()->default_value(42)
```

Другой распространенный шаблон — опция, которая может принимать несколько токенов. Такие параметры могут иметь пробелы между элементами, и они будут проанализированы в одну строку. Для этого просто используйте метод `multitoken`. Например, чтобы указать, что опция может принимать несколько значений `std::string`, используйте следующую конструкцию:

```
value<std::string>()->multitoken()
```

Если вместо этого нужно разрешить несколько экземпляров одного и того же параметра, укажите в качестве значения `std::vector`, например так:

```
value<std::vector<std::string>>()
```

Если есть логическая опция, будет использована вспомогательная функция `boost::program_options::bool_switch`, которая принимает указатель на `bool`. Если пользо-

ватель включает соответствующую опцию, функция установит для `bool`, на который направлен указатель, значение `true`. Например, следующая конструкция установит флаг `bool` с именем `flag` в `true`, если включена соответствующая опция:

```
bool_switch(&flag)
```

Класс `options_description` поддерживает `operator<<`, поэтому можно без особых усилий создать диалоговое окно справки с хорошим форматированием. В листинге 21.8 показано, как использовать ProgramOptions для создания объекта `program_options` для примера программы с именем *mgrep*.

Листинг 21.8. Использование Boost ProgramOptions для создания красиво отформатированного диалогового окна справки

```
#include <boost/program_options.hpp>
#include <iostream>
#include <string>

int main(int argc, char** argv) {
  using namespace boost::program_options;
  bool is_recursive{}, is_help{};

  options_description description{ "mgrep [options] pattern path1 path2 ..."
}; ❶
  description.add_options()
          ("help,h", bool_switch(&is_help), "display a help dialog") ❷
          ("threads,t", value<int>()->default_value(4),
                        "number of threads to use") ❸
          ("recursive,r", bool_switch(&is_recursive),
                          "search subdirectories recursively") ❹
          ("pattern", value<std::string>(), "pattern to search for") ❺
          ("paths", value<std::vector<std::string>>(), "path to search"); ❻
  std::cout << description; ❼
}
--------------------------------------------------------------------------------
mgrep [options] pattern path1 path2 ...:
-h [ --help ]             display a help dialog
-t [ --threads ] arg (=4) number of threads to use
-r [ --recursive ]        search subdirectories recursively
--pattern arg             pattern to search for
--path arg                path to search
```

Сначала с помощью пользовательской строки инициализируется объект `options_description` ❶. Затем вызывается `add_options` и начинается добавление опций: логический флаг, указывающий, отображать ли диалоговое окно справки ❷, `int`, указывающий, сколько потоков использовать ❸, другой логический флаг, указывающий, выполнять ли поиск в подкаталогах рекурсивным способом ❹, `std::string`, указывающая, какой шаблон искать в файлах ❺, и список значений `std::string`, соответствующих путям поиска ❻. Затем описание выводится в стандартный вывод ❼.

Предположим, что ваша еще не реализованная программа `mgrep` всегда будет запрашивать шаблон и аргумент `paths`. Можно преобразовать их в *позиционные аргументы*,

которые, как следует из их названия, будут назначать аргументы на основе их позиции. Для этого используется класс `boost::program_options::positional_options_description`, который не принимает аргументов конструктора. Используется метод `add`, который принимает два аргумента: строку в стиле C, соответствующую опции, которую нужно преобразовать в позиционную, и `int`, соответствующий количеству аргументов, которые нужно привязать к нему. Можно вызвать `add` несколько раз, чтобы добавить несколько позиционных аргументов. Но порядок имеет значение. Позиционные аргументы будут привязаны слева направо, поэтому первый вызов `add` применяется к левым позиционным аргументам. Для последней позиционной опции можно использовать число -1, чтобы указать `ProgramOptions` привязать все оставшиеся элементы к соответствующей опции.

В листинге 21.9 приведен фрагмент, который можно добавить в `main` листинга 21.7, чтобы добавить позиционные аргументы.

Листинг 21.9. Добавление позиционных аргументов в листинг 21.8

```
positional_options_description positional; ❶
positional.add("pattern", 1); ❷
positional.add("path", -1); ❸
```

`positional_options_description` инициализируется без параметров конструктора ❶. Затем вызывается `add` и передается аргумент `pattern` и `1`, который будет связывать первую позиционную опцию с опцией *pattern* ❷. add вызывается снова, на этот раз с передачей аргументов `path` и `-1` ❸, который будет связывать оставшиеся позиционные опции с опцией *path*.

Разбор опций

Теперь, когда вы объявили, что программа принимает параметры, можно проанализировать пользовательский ввод. Мы возьмем конфигурацию из переменных среды, файлов конфигурации и командной строки. Для краткости в этом разделе обсуждается только последнее.

ПРИМЕЧАНИЕ

Для получения информации о настройке конфигурации из переменных среды и файлов конфигурации обратитесь к документации Boost ProgramOptions, в первую очередь к руководству.

Для анализа ввода из командной строки используется класс `boost::program_options::command_line_parser`, который принимает два аргумента параметров конструктора: `int`, соответствующий `argc`, число аргументов в командной строке и `char**`, соответствующий `argv`, значению (или содержимому) аргументов в командной строке. Этот класс предлагает несколько важных методов, которые будут использоваться для объявления того, как анализатор должен интерпретировать пользовательский ввод.

Во-первых, вызывается его метод `options`, который принимает один аргумент, соответствующий параметру `options_description`. Далее будет использоваться позиционный метод, который принимает один аргумент, соответствующий `positional_options_description`. Наконец, run запустится без каких-либо аргументов. Это заставляет синтаксический анализатор анализировать ввод командной строки и возвращать объект `parsed_options`.

В листинге 21.10 приведен фрагмент, который можно добавить в `main` после листинга 21.8, чтобы включить `command_line_parser`.

Листинг 21.10. Добавление command_line_parser в листинг 21.8

```
command_line_parser parser{ argc, argv }; ❶
parser.options(description); ❷
parser.positional(positional); ❸
auto parsed_result = parser.run(); ❹
```

Инициализируется `command_line_parser` с именем `parser`, передавая аргументы из `main` ❶. Затем объект `options_description` передается методу `options` ❷, а позиция `positional_options_description` — методу `positional` ❸. Затем вызывается метод `run` для создания объекта `parsed_options` ❹.

ПРЕДУПРЕЖДЕНИЕ

Если пользователь передает входные данные, которые не анализируются, например, потому что они предоставляют опцию, не являющуюся частью описания, анализатор сгенерирует исключение, которое наследуется от `std::exception`.

Совместное использование опций и доступ к ним

Параметры программы сохраняются в классе `boost::program_options::variable_map`, который не принимает аргументов в конструкторе. Чтобы поместить проанализированные параметры в `variables_map`, используется метод `boost::program_options::store`, который в качестве первого аргумента принимает объект `parsed_options`, а объект `variable_map` — в качестве второго. Затем вызывается метод `boost::program_options::notify`, который принимает один аргумент `variable_map`. На этом этапе ваша `variables_map` содержит все опции, которые указал пользователь.

В листинге 21.11 приведен фрагмент, который можно добавить в `main` листинга 21.10, чтобы передать результаты в `variable_map`.

Листинг 21.11. Сохранение результатов в variable_map

```
variables_map vm; ❶
store(parsed_result, vm); ❷
notify(vm); ❸
```

Сначала объявляется `variables_map` ❶. Затем `parsed_result` передается из листинга 21.10 и только что объявленную `variables_map` в `store` ❷. Затем для `variables_map` вызывается `notify` ❸.

Класс `variable_map` является ассоциативным контейнером, который по сути похож на `std::map<std::string, boost::any>`. Чтобы извлечь элемент, используется `operator[]` с именем опции в качестве ключа. Результатом является `boost::any`, поэтому нужно преобразовать его в правильный тип, используя метод `as`. (Вы узнали о `boost::any` в разделе «any» на с. 453.) Очень важно проверить наличие опций, которые могут быть пустыми, с помощью метода `empty`. Если не удастся это сделать и `any` будет все равно приведен, выведется ошибка во время выполнения. В листинге 21.12 показано, как получить значения из `variable_map`.

Листинг 21.12. Получение значений из файла variable_map

```
if (is_help) std::cout << "Is help.\n"; ❶
if (is_recursive) std::cout << "Is recursive.\n"; ❷
std::cout << "Threads: " << vm["threads"].as<int>() << "\n"; ❸
if (!vm["pattern"].empty()) { ❹
  std::cout << "Pattern: " << vm["pattern"].as<std::string>() << "\n"; ❺
} else {
  std::cout << "Empty pattern.\n";
}
if (!vm["path"].empty()) { ❻
  std::cout << "Paths:\n";
  for(const auto& path : vm["path"].as<std::vector<std::string>>()) ❼
    std::cout << "\t" << path << "\n";
} else {
  std::cout << "Empty path.\n";
}
```

Поскольку используется значение `bool_switch` для опций `help` и `recursive`, вы просто напрямую используете эти логические значения, чтобы определить, запросил ли пользователь что-либо ❶ ❷. Поскольку для потоков задано значение по умолчанию, не нужно следить за тем, чтобы оно было пустым, поэтому можно извлечь его значение, используя непосредственно `<int>` ❸. Для тех опций без значений по умолчанию, как, например, шаблон, сначала проверяется `empty` ❹. Если эти параметры не пусты, можно извлечь их значения, используя `<std::string>` ❺. То же самое делается для `path` ❻, который позволяет извлекать предоставленную пользователем коллекцию как `<std::vector <std::string >>` ❼.

Соединяем все вместе

Теперь у вас есть все необходимые знания для сборки приложения на базе `Program-Options`. В листинге 21.13 показан один из способов собрать предыдущие листинги вместе.

Листинг 21.13. Полное приложение для анализа параметров командной строки с использованием предыдущих листингов

```
#include <boost/program_options.hpp>
#include <iostream>
#include <string>
```

```
int main(int argc, char** argv) {
  using namespace boost::program_options;
  bool is_recursive{}, is_help{};

  options_description description{ "mgrep [options] pattern path1 path2 ..." };
  description.add_options()
       ("help,h", bool_switch(&is_help), "display a help dialog")
       ("threads,t", value<int>()->default_value(4),
                    "number of threads to use")
       ("recursive,r", bool_switch(&is_recursive),
                     "search subdirectories recursively")
       ("pattern", value<std::string>(), "pattern to search for")
       ("path", value<std::vector<std::string>>(), "path to search");

  positional_options_description positional;
  positional.add("pattern", 1);
  positional.add("path", -1);

  command_line_parser parser{ argc, argv };
  parser.options(description);
  parser.positional(positional);

  variables_map vm;
  try {
    auto parsed_result = parser.run(); ❶
    store(parsed_result, vm);
    notify(vm);
  } catch (const std::exception& e) {
    std::cerr << e.what() << "\n";
    return -1;
  }

  if (is_help) { ❷
    std::cout << description;
    return 0;
  }
  if (vm["pattern"].empty()) { ❸
    std::cerr << "You must provide a pattern.\n";
    return -1;
  }
  if (vm["path"].empty()) { ❹
    std::cerr << "You must provide at least one path.\n";
    return -1;
  }
  const auto threads = vm["threads"].as<int>();
  const auto& pattern = vm["pattern"].as<std::string>();
  const auto& paths = vm["path"].as<std::vector<std::string>>();
  // Продолжение программы ... ❺
  std::cout << "Ok." << std::endl;
}
```

Первое отклонение от предыдущих списков состоит в том, что вызов для запуска оборачивается в парсере с помощью блока `try-catch`, чтобы уменьшить ошибочный ввод, предоставленный пользователем ❶. Если предоставлен ошибочный ввод, просто перехватывается исключение, выводится ошибка в стандартный вывод ошибок и вызывается `return`.

После объявления параметров программы и сохранения их, как в листингах с 21.8 по 21.12, сначала проверяется, запросил ли пользователь справочный диалог ❷. Если это так, просто выводится использование и работа завершается, так как необходимости выполнять какую-либо дополнительную проверку нет. Затем выполняется некоторая проверка ошибок, чтобы убедиться, что пользователь указал шаблон ❸ и хотя бы один путь ❹. Если нет, выводится ошибка с правильным использованием программы и работа программы завершается; в противном случае можно продолжить писать программу ❺.

В листинге 21.14 показаны различные выходные данные программы, которые скомпилированы в двоичный файл `mgrep`.

Листинг 21.14. Различные вызовы и выходные данные программы из листинга 21.13

```
$ ./mgrep ❶
You must provide a pattern.
$ ./mgrep needle ❷
You must provide at least one path.
$ ./mgrep --supercharge needle haystack1.txt haystack2.txt ❸
unrecognised option '--supercharge'
$ ./mgrep --help ❹
mgrep [options] pattern path1 path2 ...:
  -h [ --help ]             display a help dialog
  -t [ --threads ] arg (=4) number of threads to use
  -r [ --recursive ]        search subdirectories recursively
  --pattern arg             pattern to search for
  --path arg                path to search
$ ./mgrep needle haystack1.txt haystack2.txt haystack3.txt ❺
Ok.
$ ./mgrep --recursive needle haystack1.txt ❻
Ok.
$ ./mgrep -rt 10 needle haystack1.txt haystack2.txt ❼
Ok.
```

Первые три вызова возвращают ошибки по разным причинам: не указан шаблон ❶, не указан путь ❷ или указан нераспознанный параметр ❸.

В следующем вызове вы получите дружественный справочный диалог, потому что была указана опция `--help` ❹. Последние три вызова правильно анализируются, поскольку все они содержат шаблон и хотя бы один путь. Первый не содержит опций ❺, второй использует обычный синтаксис опции ❻, а третий использует сокращенный синтаксис опции ❼.

Отдельные моменты компиляции

В этом разделе объясняются несколько важных функций препроцессора, которые помогут понять проблему двойного включения, описанную в следующем подразделе, а также способы ее решения. Вы узнаете о различных вариантах оптимизации кода с помощью флагов компилятора. Кроме того, вы узнаете, как разрешить компоновщику взаимодействовать с C, используя специальное ключевое слово языка.

Пересмотр препроцессора

Препроцессор — это программа, которая применяет простые преобразования к исходному коду перед компиляцией. Вы даете инструкции препроцессору с помощью директив препроцессора. Все директивы препроцессора начинаются с хеш-метки (`#`). Вспомните из «Цепочки инструментов компилятора» на с. 61, что `#include` — это директива препроцессора, которая указывает препроцессору копировать и вставлять содержимое соответствующего заголовка непосредственно в исходный код.

Препроцессор также поддерживает другие директивы. Наиболее распространенным является *макрос*, который является именованым фрагментом кода. Всякий раз, когда это имя используется в коде C++, препроцессор заменяет его содержимым макроса.

Два разных типа макросов являются объектно-подобными и функционально-подобными. Объекто-подобный макрос объявляется, используя следующий синтаксис:

```
#define <NAME> <CODE>,
```

где `NAME` — имя макроса, а `CODE` — код для замены этого имени.

Например, в листинге 21.15 показано, как определить строковый литерал для макроса.

Листинг 21.15. Программа на C++ с объектно-подобным макросом

```
#include <cstdio>
#define MESSAGE "LOL" ❶

int main(){
  printf(MESSAGE); ❷
}
--------------------------------------------------------------------------------
LOL
```

Определяется макрос `MESSAGE` для соответствия с кодом `"LOL"` ❶. Затем используется макрос `MESSAGE` в качестве строки формата для `printf` ❷. После того как препроцессор завершил работу над листингом 21.15, он отображается как код из листинга 21.16 для компилятора.

Листинг 21.16. Результат препроцессинга листинга 21.15

```
#include <cstdio>

int main(){
  printf("LOL");
}
```

Препроцессор здесь — не что иное, как инструмент копирования и вставки. Макрос исчезает, и остается простая программа, которая выводит `LOL` в консоль.

ПРИМЕЧАНИЕ

У компиляторов обычно есть флаг, который ограничивает компиляцию только шагом препроцессинга, что позволяет проверить работу, которую выполняет препроцессор. Это заставит компилятор выдавать предварительно обработанный исходный файл, соответствующий каждой единице трансляции. В GCC, Clang и MSVC, например, можно использовать флаг `-E`.

Функционально-подобный макрос похож на объектный макрос, за исключением того, что он может принимать список параметров после своего идентификатора:

```
#define <NAME>(<PARAMETERS>) <CODE>
```

Можно использовать эти `PARAMETERS` в `CODE`, что позволяет пользователю настраивать поведение макроса. Листинг 21.17 содержит функционально-подобный макрос `SAY_LOL_WITH`.

Листинг 21.17. Программа на C++ с функционально-подобным макросом

```
#include <cstdio>
#define SAY_LOL_WITH(fn) fn("LOL") ❶

int main() {
  SAY_LOL_WITH(printf); ❷
}
```

Макрос `SAY_LOL_WITH` принимает один параметр с именем `fn` ❶. Препроцессор вставляет макрос в выражение `fn("LOL")`. Когда он вычисляет `SAY_LOL_WITH`, препроцессор вставляет `printf` в выражение ❷, получая единицу трансляции, как показано в листинге 21.16.

Условная компиляция

Препроцессор также предлагает *условную компиляцию*, средство, которое обеспечивает базовую логику `if-else`. Доступно несколько вариантов условной компиляции, но тот, с которым вы, вероятно, столкнетесь, показан в листинге 21.18.

Если `MY_MACRO` не определен в точке, где препроцессор оценивает `#ifndef` ❶, листинг 21.18 сводится к коду, представленному `// Сегментом 1` ❷. Если `MY_MACRO` `#defined`, в листинге 21.18 вычисляется код, представленный `// Сегментом 2` ❸, `#else` не является обязательным.

Листинг 21.18. Программа на C++ с условной компиляцией

```
#ifndef MY_MACRO ❶
// Сегмент 1 ❷
#else
// Сегмент 2 ❸
#endif
```

Двойное включение

Помимо использования `#include`, использовать препроцессор следует как можно реже. Препроцессор чрезвычайно примитивен и вызовет ошибки при отладке, если вы слишком сильно рассчитываете на него. Это очевидно с `#include`, который является простой командой копирования и вставки.

Поскольку можно определить символ только один раз (здесь действует *правило единого определения*), стоит убедиться, что ваши заголовки не пытаются переопределить символы. Самый простой способ сделать эту ошибку — дважды включить один и тот же заголовок. Это называется *проблемой двойного включения*.

Обычный способ избежать проблемы двойного включения — использовать условную компиляцию для добавления *предохранителя включения*. Предохранитель включения определяет, был ли заголовок включен ранее. Если это так, он использует условную компиляцию для очистки заголовка. В листинге 21.19 показано, как поместить защитные элементы вокруг заголовка.

Листинг 21.19. Файл step_function.h, обновленный включенными охранниками

```
// step_function.h
#ifndef STEP_FUNCTION_H ❶
int step_function(int x);
#define STEP_FUNCTION_H ❷
#endif
```

При первом включении препроцессором `step_function.h` в исходный файл макрос `STEP_FUNCTION_H` не будет определен, поэтому `#ifndef` ❶ возвращает код вплоть до `#endif`. В этом коде определяется (`#define`) макрос `STEP_FUNCTION_H` ❷. Это гарантирует, что, если препроцессор снова включит `step_function.h`, `#ifndef STEP_FUNCTION_H` будет иметь значение `false` и код не будет сгенерирован.

Предохранители включения настолько распространены, что большинство современных цепочек инструментов поддерживают специальный синтаксис `#pragmaonce`. Если один из поддерживающих препроцессоров видит эту строку, то будет вести себя так, как будто в заголовке есть предохранители включения. Это устраняет лишние церемонии. Используя эту конструкцию, можно преобразовать листинг 21.19 в листинг 21.20.

Листинг 21.20. step_function.h, обновленный с помощью #pragma once

```
#pragma once ❶
int step_function(int x);
```

Все, что вы здесь сделали, — запустили заголовок с `#pragma once` ❶, что является предпочтительным методом. Как правило, каждый заголовок стоит начинать с `#pragma once`.

Оптимизация компилятора

Современные компиляторы могут выполнять сложные преобразования кода для увеличения производительности во время выполнения и уменьшения размера двоичного файла. Эти преобразования называются *оптимизациями* и влекут за собой определенные затраты для программистов. Оптимизация обязательно увеличивает время компиляции. Кроме того, оптимизированный код зачастую сложнее отлаживать, чем неоптимизированный, поскольку оптимизатор обычно исключает и переупорядочивает инструкции. Короче говоря, обычно нужно отключить оптимизации во время программирования, но включить их во время тестирования и внедрения. Соответственно, компиляторы обычно предоставляют несколько вариантов оптимизации. В таблице 21.1 описывается один такой пример — варианты оптимизации, доступные в GCC 8.3, хотя эти флаги довольно распространены среди основных компиляторов.

Таблица 21.1. Варианты оптимизации в GCC 8.3

Флаг	Описание
`-O0` (по умолчанию)	Сокращает время компиляции, отключая оптимизации. Дает хороший опыт отладки, но неоптимальную производительность во время выполнения
`-O` или `-O1`	Выполняет большинство доступных оптимизаций, но пропускает те, которые могут занимать много времени (компиляции)
`-O2`	Выполняет все оптимизации при `-O1`, а также почти все оптимизации, которые существенно не увеличивают размер двоичного файла. Компиляция может занять гораздо больше времени, чем с `-O1`
`-O3`	Выполняет все оптимизации при `-O2`, а также множество оптимизаций, которые могут существенно увеличить размер двоичного файла. Опять же это увеличивает время компиляции по сравнению с `-O1` и `-O2`
`-Os`	Оптимизирует аналогично `-O2`, но с приоритетом уменьшения двоичного размера. Можно представить это как связь с `-O3`, которая готова увеличить размер двоичного файла в обмен на производительность. Любые оптимизации `-O2`, которые не увеличивают размер двоичного файла, выполняются
`-Ofast`	Включает все оптимизации `-O3`, а также некоторые опасные оптимизации, которые могут нарушать соответствие стандартам. Пусть покупатель будет бдителен
`-Og`	Включает оптимизации, которые не ухудшают возможности отладки. Обеспечивает хороший баланс разумных оптимизаций, быстрой компиляции и простоты отладки

В обычных случаях используйте `-O2` для бинарного файла, если веских причин для его изменения нет. Для отладки используйте `-Og`.

Связь с C

Можно позволить коду C включать функции и переменные из ваших программ, используя *языковые связи*. Языковая связь инструктирует компилятор генерировать символы с определенным форматом, дружественным к другому целевому языку. Например, чтобы позволить программе на C использовать ваши функции, просто добавьте в свой код языковую связь `extern "C"`.

Рассмотрим заголовок `sum.h` из листинга 21.21, который генерирует C-совместимый символ для `sum`.

Листинг 21.21. Заголовок, который делает функцию sum доступной для редакторов связей C

```
 // sum.h
#pragma once
extern "C" int sum(const int* x, int len);
```

Теперь компилятор будет генерировать объекты, которые может использовать редактор связей C. Чтобы использовать эту функцию в коде C, просто объявите функцию `sum` как обычно:

```
int sum(const int* x, size_t len);
```

Затем дайте команду редактору связей C включить объектный файл C++.

ПРИМЕЧАНИЕ

В соответствии со стандартом C++, прагма — это метод для предоставления компилятору дополнительной информации помимо того, что встроено в исходный код. Эта информация определяется реализацией, поэтому компилятору не требуется каким-либо образом использовать информацию, указанную в прагме. «Прагма» — это греческий корень слова «факт».

Также можно взаимодействовать противоположным образом: использовать вывод компилятора C в программах на C++, предоставив редактору связей объектный файл, сгенерированный компилятором C.

Предположим, что компилятор C сгенерировал функцию, эквивалентную `sum`. Можно выполнить компиляцию, используя заголовок `sum.h`, и редактор связей не будет иметь проблем с использованием объектного файла, благодаря языковой связи.

Если внешних функций много, можно использовать фигурные скобки `{}`, как показано в листинге 21.22.

Листинг 21.22. Рефакторинг листинга 21.21, содержащий несколько функций с модификатором extern

```
// sum.h
#pragma once

extern "C" {
  int sum_int(const int* x, int len);
  double sum_double(const double* x, int len);
--пропуск--
}
```

Функции `sum_int` и `sum_double` будут иметь связь с языком C.

ПРИМЕЧАНИЕ

Взаимодействие между `C++` и `Python` можно осуществлять с помощью `Boost Python`. Подробности см. в документации `Boost`.

Итоги

В этой главе вы узнали о функциях поддержки программ, которые позволяют работать с жизненным циклом приложения. Далее изучили Boost ProgramOptions, который позволяет легко принимать ввод от пользователей с помощью декларативного синтаксиса. Затем рассмотрели некоторые особенности компиляции, которые будут полезны при расширении горизонтов разработки приложений на C++.

Упражнения

21.1. Добавьте изящную обработку прерываний с клавиатуры к асинхронному эхо-серверу верхнего регистра в листинге 20.12. Добавьте переключатель уничтожения со статической длительностью хранения, который объекты и приемники сеанса проверяют перед постановкой в очередь более асинхронного ввода-вывода.

21.2. Добавьте параметры программы к асинхронному HTTP-клиенту в листинге 20.10. Он должен принимать параметры для хоста (например, www.nostarch.com) и одного или нескольких ресурсов (например, /index.htm). Следует создать отдельный запрос для каждого ресурса.

21.3. Добавьте еще одну опцию к программе в упражнении 21.2, которая принимает каталог, в который вы будете записывать все ответы HTTP. Получите имя файла от каждой комбинации хоста/ресурса.

21.4. Реализуйте программу mgrep. Она должна включать в себя многие библиотеки, о которых вы узнали в части II. Исследуйте алгоритм поиска Бойера – Мура в Boost Algorithm (в заголовке `<boost/algorithm/search/boyer_moore.hpp>`). Используйте `std::async` для запуска задач и определения способа координации работы между ними.

Что еще почитать?

- «The Boost C++ Libraries», 2nd Edition, Boris Schäling (XML Press, 2014)
- API Design for C++ by Martin Reddy (Morgan Kaufmann, 2011)

Джош Лоспинозо

C++ для профи

Перевел с английского С. Черников

Заведующая редакцией	*Ю. Сергиенко*
Ведущий редактор	*К. Тульцева*
Корректоры	*Г. Шкатова, М. Одинокова*
Обложка	*В. Мостипан*
Верстка	*Е. Неволайнен*

Изготовлено в России. Изготовитель: ООО «Прогресс книга».
Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург,
Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.
Дата изготовления: 02.2021. Наименование: книжная продукция.
Срок годности: не ограничен.
Налоговая льгота — общероссийский классификатор продукции ОК 034-2014,
58.11.12 — Книги печатные профессиональные, технические и научные.
Импортер в Беларусь: ООО «ПИТЕР М», 220020,
РБ, г. Минск, ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.
Подписано в печать 25.01.21. Формат 70x100/16. Бумага офсетная. Усл. п. л. 67,080. Тираж 900. Заказ