Современный C++ далеко ушел после 2011 года. Последнее обновление стандарта — C++17 — уже утверждено и внедряется в некоторые реализации.
Эта книга послужит вам путеводителем по стандартной библиотеке C++ и познакомит с самыми новыми возможностями, включенными в C++17.

Издание начинается с подробного исследования стандартной библиотеки шаблонов C++STL (Standard Template Library). Вы узнаете, чем отличаются классический полиморфизм от обобщенного программирования, лежащего в основе STL. Также вы увидите, как использовать на практике разные алгоритмы и контейнеры, имеющиеся в STL. Далее следует описание инструментов современного C++. В этой части вы познакомитесь с алгебраическими типами, такими как std:: optional, словарными типами, такими как std:: function, умными указателями и примитивами синхронизации, такими как std:: atomic и std:: mutex. В заключительной части вашему вниманию будет представлена поддержка регулярных выражений в C++ и операций ввода/вывода с файлами.

К концу книги вы получите достаточно полное представление о возможностях и внутренних механизмах стандартной библиотеки C++17, чтобы использовать их в своих программах и библиотеках.

С этой книгой вы:

- научитесь создавать свои типы итераторов, диспетчеров памяти и пулов потоков выполнения;
- овладеете стандартными контейнерами и стандартными алгоритмами;
- усовершенствуете свой код, заменив new/delete умными указателями;
- усвоите разницу между мономорфными, полиморфными и обобщенными алгоритмами;
- узнаете смысл и назначение словарных типов, типов-произведений и типов-сумм.

Осваиваем C++17 STL

Артур О'Двайр

Осваиваем C++17 STL

Используйте компоненты стандартной библиотеки C++17 в полной мере

Интернет -магазин:
www.dmkpress.com
Книга - почтой:
e-mail: orders@alians-kniga.ru
Оптовая продажа:
«Альянс-книга»
Тел/факс (499)782-3889
e-mail: books@alians-kniga.ru

Packt

ISBN 978-5-97060-663-6

Артур О'Двайр

Осваиваем C++17 STL

Используйте компоненты стандартной библиотеки в C++17 в полной мере

Mastering the C++17 STL

Make full use of the standard library components in C++17

Arthur O'Dwyer

BIRMINGHAM - MUMBAI

Осваиваем C++17 STL

Используйте компоненты стандартной библиотеки в C++17 в полной мере

Артур О'Двайр

Москва, 2019

УДК 004.4
ББК 32.973.202-018.2
Д22

Д22 Артур О'Двайр
Осваиваем C++17 STL / пер. с англ. А. Н. Киселева. – М.: ДМК Пресс, 2019. – 352 с.: ил.

ISBN 978-5-97060-663-6

Стандарт C++17, которому посвящена книга, удвоил объем библиотеки в сравнении с C++11. Вы узнаете о наиболее важных особенностях стандартной библиотеки C++17 со множеством примеров, научитесь создавать свои типы итераторов, диспетчеры памяти, пулы потоков выполнения. Также рассмотрены отличия мономорфизма, полиморфизма и обобщенных алгоритмов.

Издание адресовано разработчикам, желающим овладеть новыми особенностями библиотеки C++17 STL и в полной мере использовать ее компоненты. Знакомство с языком C++ является обязательным условием.

УДК 004.4
ББК 32.973.202-018.2

ISBN 978-1-78712-682-4 (англ.)
ISBN 978-5-97060-663-6 (рус.)

Оглавление

Предисловие от разработчиков C++ в России и Беларуси

Совсем немного языков могут похвастаться такой богатой и яркой историей, как язык C++. Уже несколько десятилетий C++ является де-факто стандартом для реализации такого сложнейшего программного обеспечения, как операционные системы, базы данных, компиляторы, веб-браузеры, системы защиты данных и антивирусы, микроконтроллеры; военная промышленность, космос, высоконагруженные системы и множество других доменов немыслимы без использования этой технологии. Огромные базы исходных кодов, созданные за эти годы, требуют поддержки и развития, что немыслимо без стандартизации. Закон иерархических компенсаций профессора Седова в формулировке профессора Назаретяна гласит, что многообразие на верхнем уровне иерархии систем возможно только при условии стандартизации нижележащих уровней иерархии. У каждого на слуху интернет вещей, искусственный интеллект, обработка больших данных, умные автомобили, включая беспилотные, – эти области и являются вершиной ИТ-иерархии сегодня. Ядро большинства перечисленных трендовых направлений разрабатывается с использованием языка C++, а значит, согласно закону Седова, стандартизация C++ является необходимым условием для самого существования современных ИТ-трендов. Успехи ИТ-сферы сегодняшнего дня обусловлены в том числе преодолением «кризиса» стандартизации языка в период от C++98/03 к C++11. Сейчас комитет стандартизации выпускает новые версии языка в среднем раз в три года – C++11, 14, 17. Ведется огромная исследовательская работа ведущих ученых и программистов мира по подготовке нового стандарта C++20. Такая регулярность является замечательным показателем того, что ниша языка неуклонно растет! Освоение последних стандартов – необходимое условие для успешного профессионального роста и самореализации в рамках современных ИТ-векторов. Исторически наша страна обладает огромной научно-исследовательской базой, основанной на традициях быстрого изучения новых, только зарождающихся дисциплин. Сейчас у отечественных специалистов появился уникальный шанс значительно увеличить присутствие в нише C++-разработки. Новые стандарты – это новые возможности для самореализации и рывка вперед отечественной ИТ-отрасли.

Изучение C++ – крайне трудоемкий процесс, который проходит гораздо легче и продуктивнее при общении специалистов друг с другом. Профессиональные группы и сообщества – неотъемлемый атрибут этого процесса. Автор книги Артур О'Двайр, являясь активным участником комитета стандартизации ISO C++, одновременно известен организацией встреч пользователей C++ в Заливе Сан-Франциско. Также и в нашей стране во многих крупных городах проходят регулярные встречи местных C++-сообществ, на которых есть возможность обсудить с коллегами нюансы новых стандартов, выступить с докладом, завязать профессиональные контакты и даже поучаствовать в стандартизации языка. Давайте же читать книги, осваивать новые горизонты языка C++, общаться, развиваться и расти профессионально вместе, двигая ИТ-индустрию вперед!

Антон Наумович и Антон Семенченко
Сообщество C++-разработчиков CoreHard
cpp-russia.ru, stdcpp.ru, corehard.by

Об авторе

Артур О'Двайр (Arthur O'Dwyer) использует современный язык C++ в своей повседневной работе около десяти лет – еще с тех пор, когда под «современным C++» подразумевался «классический C++». С 2006 по 2011 год он принимал участие в работе над компилятором Green Hills C++ Compiler. Начиная с 2014 г. организовывал еженедельные встречи пользователей C++ в Заливе Сан-Франциско и регулярно выступает, освещая темы, которые можно найти в этой книге. В 2018 году он во второй раз присутствовал на заседании комитета ISO C++.

Это его первая книга.

О научном редакторе

Уилл Бреннан (Will Brennan) – разработчик программ на C++/Python. Живет в Лондоне и имеет богатый опыт высокопроизводительных приложений обработки изображений и машинного обучения. Вы можете найти его на GitHub: https://github.com/WillBrennan.

Предисловие

Язык программирования C++ имеет долгую историю, уходящую корнями в 1980-е. Недавно он пережил возрождение благодаря появлению новых возможностей в стандартах 2011 и 2014 годов. Пока книга готовилась к печати, вышел новый стандарт C++17.

Стандарт C++11 практически удвоил объем стандартной библиотеки, добавив такие заголовки, как `<tuple>`, `<type_traits>` и `<regex>`. Стандарт C++17 снова удвоил объем библиотеки, добавив новые заголовки, такие как `<optional>`, `<any>` и `<filesystem>`. Программист, много времени тративший на разработку кода и не следивший за процессом стандартизации, вполне мог почувствовать себя отставшим от жизни – в стандартной библиотеке появилось так много нового, что он никогда не смог бы овладеть всеми нововведениями в одиночку или хотя бы отделить зерна от плевел. В конце концов, кому захочется провести месяц, читая техническую документацию о `std::locale` и `std::ratio`, только чтобы узнать, что в них нет ничего, что могло бы пригодиться ему в повседневной работе?

В этой книге я расскажу вам о наиболее важных особенностях стандартной библиотеки C++17. Для краткости я опущу некоторые разделы, такие как вышеупомянутый `<type_traits>`; но мы рассмотрим всю современную стандартную библиотеку шаблонов STL (каждый стандартный контейнер и каждый стандартный алгоритм), плюс такие важные темы, как умные указатели, случайные числа, регулярные выражения и новую для C++17 библиотеку `<filesystem>`.

Я буду знакомить вас с новинками на примерах. Вы научитесь создавать свои типы итераторов; свои диспетчеры памяти на основе `std::pmr::memory_resource`; свои пулы потоков выполнения с использованием `std::future`.

Я расскажу об идеях, которые вы не найдете в справочных руководствах. Вы узнаете, чем отличаются мономорфизм, полиморфизм и обобщенные алгоритмы (глава 1 «*Классический полиморфизм и обобщенное программирование*»); что означает для `std::string` или `std::any` быть «словарным типом» (глава 5 «*Словарные типы*») и чего можно ожидать от грядущего стандарта C++20 и потом.

Я предполагаю, что вы уже достаточно хорошо знакомы с основами языка C++11; например, что вы уже понимаете, как писать шаблонные классы и функции, знаете, чем отличаются ссылки *lvalue* и *rvalue*, и т. д.

Содержание книги

Глава 1 «*Классический полиморфизм и обобщенное программирование*» охватывает классический полиморфизм (виртуальные функции-члены) и обобщенное программирование (шаблоны).

Глава 2 «*Итераторы и диапазоны*» объясняет идею представления итератора как обобщенного указателя и практическую пользу полуоткрытых диапазонов, выраженных в виде пары итераторов.

Глава 3 «Алгоритмы с парами итераторов» исследует широкое разнообразие обобщенных алгоритмов, оперирующих диапазонами, выраженными в виде пар итераторов.

Глава 4 «Зоопарк контейнеров» исследует не менее широкое разнообразие стандартных шаблонных классов контейнеров и рассказывает, какие контейнеры лучше подходят для тех или иных задач.

Глава 5 «Словарные типы» проведет вас через царство алгебраических типов, таких как `std::optional`, и ABI-совместимые стираемые типы, такие как `std::function`.

Глава 6 «Умные указатели» рассказывает о назначении и особенностях использования умных указателей.

Глава 7 «Конкуренция» охватывает атомы, мьютексы, условные переменные, потоки выполнения, объекты `future` и `promise`.

Глава 8 «Диспетчеры памяти» описывает новый заголовок `<memory_resource>`, появившийся в C++17.

Глава 9 «Потоки ввода/вывода» исследует развитие модели ввода/вывода в C++, от `<unistd.h>` до `<stdio.h>` и `<iostream>`.

Глава 10 «Регулярные выражения» научит вас пользоваться регулярными выражениями в C++.

Глава 11 «Случайные числа» описывает поддержку генераторов псевдослучайных чисел в C++.

Глава 12 «Файловая система» охватывает новую библиотеку `<filesystem>`, появившуюся в C++17.

Что потребуется для работы с книгой

Поскольку эта книга не является справочным руководством, вам может пригодиться такое руководство, как cppreference (en.cppreference.com/w/cpp), где вы сможете прояснить любые детали. Вам также определенно понадобится компилятор, поддерживающий стандарт C++17. На момент подготовки книги к печати существовало несколько компиляторов с более или менее полноценной поддержкой C++17, включая GCC, Clang и Microsoft Visual Studio. Вы можете установить их у себя локально или воспользоваться многочисленными бесплатными онлайн-службами, такими как Wandbox (wandbox.org), Godbolt (gcc.godbolt.org) и Rextester (rextester.com).

Кому адресована эта книга

Эта книга адресована разработчикам, желающим овладеть новыми особенностями библиотеки C++17 STL и в полной мере использовать ее компоненты. Знакомство с языком C++ является обязательным условием.

Типографские соглашения

В этой книге используется несколько разных стилей оформления текста с целью обеспечить визуальное отличие информации разных типов. Ниже приводится несколько примеров таких стилей оформления и краткое описание их назначения.

Программный код в тексте, имена таблиц баз данных, имена папок, имена файлов, расширения файлов, пути в файловой системе, адреса URL, ввод пользователя и ссылки в Twitter оформляются, как показано в следующем предложении: «Функция `buffer()` принимает аргументы типа `int`».

Блоки программного кода оформляются так:

```
try {
  none.get();
} catch (const std::future_error& ex) {
  assert(ex.code() == std::future_errc::broken_promise);
}
```

Новые термины и **важные определения** будут выделяться в обычном тексте жирным.

Таким значком будут оформляться предупреждения и важные примечания.

Таким значком будут оформляться советы и рекомендации.

Отзывы и пожелания

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

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

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

Скачивание исходного кода примеров

Скачать файлы с дополнительной информацией для книг издательства «ДМК Пресс» можно на сайте www.dmkpress.com или www.дмк.рф в разделе «Читателям – Файлы к книгам».

Кроме того, примеры кода к книге доступны на сайте GitHub, по адресу: https://github.com/PacktPublishing/ Mastering-the-Cpp17-STL.

Список опечаток

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

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

Нарушение авторских прав

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

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

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

Глава 1

Классический полиморфизм и обобщенное программирование

Стандартная библиотека C++ преследует две разные, но одинаково важные цели. Первая цель – предоставить надежные реализации некоторых конкретных типов данных или функций, которые могут пригодиться в разных программах, но не являются частью базового синтаксиса языка. Именно поэтому стандартная библиотека включает `std::string`, `std::regex`, `std::filesystem::exists` и т. д. Другая цель – предоставить надежные реализации широко используемых абстрактных алгоритмов сортировки, поиска, обращения, сравнения и т. д. В этой главе мы узнаем, что подразумевается под словами «абстрактный код», и рассмотрим два подхода к определению абстрактного кода, которые используются в стандартной библиотеке: *классический полиморфизм* и *обобщенное программирование*.

В этой главе рассматриваются следующие темы:

- конкретные (мономорфные) функции, поведение которых не параметризуется;
- классический полиморфизм: базовые классы, виртуальные функции-члены и наследование;
- обобщенное программирование: концепции, требования и модели;
- практические достоинства и недостатки каждого из подходов.

Конкретные мономорфные функции

Что отличает абстрактный алгоритм от конкретной функции? Ответить на этот вопрос нам поможет пример. Напишем функцию, умножающую каждый элемент массива на 2:

```
class array_of_ints {
  int data[10] = {};
```

```
public:
  int size() const { return 10; }
  int& at(int i) { return data[i]; }
};

void double_each_element(array_of_ints& arr)
{
  for (int i=0; i < arr.size(); ++i) {
    arr.at(i) *= 2;
  }
}
```


Наша функция `double_each_element` работает *только* с объектами типа `array_of_int`; попытка передать объект другого типа не увенчается успехом (код просто не будет компилироваться). Функции, такие как наша `double_each_element`, называют *конкретными*, или *мономорфными*. Мы называем их *конкретными*, потому что они недостаточно *абстрактны* для наших целей. Просто представьте, насколько неудобно было бы, если бы стандартная библиотека C++ предоставляла конкретную процедуру `sort`, работающую только с определенным типом данных!

Классические полиморфные функции

Мы можем повысить уровень абстракции наших алгоритмов, применив приемы классического **объектно-ориентированного** (ОО) программирования, широко используемые в таких языках, как Java и C#. Суть подхода ОО состоит в том, чтобы решить, какие варианты поведения должны быть настраиваемыми, и затем объявить их в виде общедоступных виртуальных функций-членов *абстрактного базового класса*:

```
class container_of_ints {
public:
  virtual int size() const = 0;
  virtual int& at(int) = 0;
};

class array_of_ints : public container_of_ints {
  int data[10] = {};
public:
  int size() const override { return 10; }
  int& at(int i) override { return data[i]; }
};

class list_of_ints : public container_of_ints {
  struct node {
    int data;
    node *next;
  };
```

```
  node *head_ = nullptr;
  int size_ = 0;
public:
  int size() const override { return size_; }
  int& at(int i) override {
    if (i >= size_) throw std::out_of_range("at");
    node *p = head_;
    for (int j=0; j < i; ++j) {
      p = p->next;
    }
    return p->data;
  }
  ~list_of_ints();
};

void double_each_element(container_of_ints& arr)
{
  for (int i=0; i < arr.size(); ++i) {
    arr.at(i) *= 2;
  }
}

void test()
{
  array_of_ints arr;
  double_each_element(arr);

  list_of_ints lst;
  double_each_element(lst);
}
```

Два разных вызова `double_each_element` в `test` успешно компилируются, потому что с точки зрения классического ОО `array_of_ints` **ЯВЛЯЕТСЯ** `container_of_ints` (то есть наследует `container_of_ints` и реализует соответствующие виртуальные функции-члены) и `list_of_ints` также **ЯВЛЯЕТСЯ** `container_of_ints`. Однако поведение любого данного объекта `container_of_ints` параметризуется его *динамическим типом*; то есть определяется таблицей указателей на функции, связанной с конкретным объектом.

Поскольку теперь поведение функции `double_each_element` можно параметризовать, не меняя ее исходный код, а просто передавая ей объекты разных динамических типов, мы говорим, что функция является *полиморфной*.

Но такая полиморфная функция может работать только с типами, которые наследуют базовый класс `container_of_ints`. Например, у вас не получится передать этой функции вектор `std::vector<int>`; при попытке сделать это вы получите ошибку компиляции. Классический полиморфизм – удобный прием, но не дает нам полной обобщенности.

Преимущество классического (объектно-ориентированного) полиморфизма в том, что исходный код все еще однозначно соответствует машинному

коду, генерируемому компилятором. На уровне машинного кода у нас имеется только одна функция `double_each_element`, с одной сигнатурой и с одной, четко определенной точкой входа. Например, мы легко можем получить адрес функции `double_each_element` и сохранить его в указателе.

Обобщенное программирование с шаблонами

В современном C++ полностью обобщенные алгоритмы обычно записываются в виде *шаблонов*. Мы все еще должны реализовать шаблонную функцию в терминах общедоступных функций-членов `.size()` и `.at()`, но при этом отпадает требование к принадлежности аргумента `arr` определенному типу. Поскольку наша новая функция будет шаблонной, мы должны сообщить компилятору, что нас не интересует конкретный тип `arr`, что для каждого нового типа `arr` он должен сгенерировать совершенно новую функцию (то есть создать экземпляр шаблона) с параметром этого типа.


```
template<class ContainerModel>
void double_each_element(ContainerModel& arr)
{
  for (int i=0; i < arr.size(); ++i) {
    arr.at(i) *= 2;
  }
}

void test()
{
  array_of_ints arr;

  double_each_element(arr);

  list_of_ints lst;
  double_each_element(lst);

  std::vector<int> vec = {1, 2, 3};
  double_each_element(vec);
}
```

В большинстве случаев возможность точно выразить словами, какие операции должны поддерживаться шаблонным параметром типа `ContainerModel`, помогает создавать более совершенные программы. Такой набор операций определяет то, что в C++ называют *концепцией*; в этом примере мы можем сказать, что концепция `Container` заключается в «наличии функции-члена с именем `size`, которая возвращает размер контейнера в виде значения типа `int` (или другого, сравнимого с `int`); и в наличии функции-члена с именем `at`, которая принимает индекс типа `int` (или другого, который неявно может преобразовываться в тип `int`) и возвращает неконстантную ссылку на элемент в контейнере с этим *индексом*». Если некоторый класс `array_of_ints`

поддерживает операции, требуемые концепцией `Container`, так что объекты этого класса могут передаваться в вызов `double_each_element`, мы говорим, что конкретный класс `array_of_ints` является *моделью* концепции `Container`. Именно поэтому я дал имя `ContainerModel` параметру типа шаблона в предыдущем примере.

Более традиционно было бы использовать имя `Container` для параметра типа шаблона, и с этого момента я так и буду поступать; просто я не хотел начинать с путаницы между концепцией `Container` и конкретным параметром типа шаблона в этой конкретной шаблонной функции, которая выглядит так, будто ее аргумент принадлежит конкретному классу, моделирующему концепцию `Container`.

Реализация абстрактного алгоритма с использованием шаблонов, когда поведение алгоритма параметризуется на этапе компиляции типами, моделирующими соответствующие концепции, называется обобщенным программированием.

Обратите внимание, что в нашем описании концепции `Container` нигде не говорится о том, что элементы контейнера должны иметь тип `int`; и не случайно мы теперь можем использовать нашу обобщенную функцию `double_each_element` даже с контейнерами, содержащими значения других типов, отличных от `int`!

```
std::vector<double> vecd = {1.0, 2.0, 3.0};
double_each_element(vecd);
```

Этот дополнительный уровень обобщенности является одним из важнейших достоинств шаблонов C++ в обобщенном программировании, в отличие от классического полиморфизма. Классический полиморфизм скрывает разность поведений разных классов за неизменной сигнатурой (например, `.at(i)` всегда возвращает `int&`), но, как только появляется необходимость использовать разные сигнатуры, классический полиморфизм перестает быть хорошим инструментом для этой работы.

Другое достоинство обобщенного программирования – высокая скорость благодаря расширенным возможностям встраивания. Пример, реализованный в стиле классического полиморфизма, вынужден снова и снова обращаться к таблице виртуальных методов объекта `container_of_int`, чтобы получить адрес конкретного виртуального метода `at`, и, как правило, лишен возможности миновать этот поиск на этапе компиляции. Шаблонная функция `double_each_element<array_of_int>`, напротив, может скомпилировать непосредственный вызов `array_of_int::at` или даже встроить тело этой функции в код.

Поскольку обобщенное программирование с шаблонами легко справляется со сложными требованиями и обеспечивает гибкую поддержку разных типов – даже таких простых, как `int`, где классический полиморфизм терпит неудачу, – стандартная библиотека использует шаблоны для всех алгоритмов в ней и контейнеров, которыми эти алгоритмы оперируют. По этой причине часть стандартной библиотеки, имеющей отношение к алгоритмам и контейнерам, часто называют *стандартной библиотекой шаблонов* (Standard Template Library, STL).

Все верно – технически STL является лишь малой частью стандартной библиотеки C++! Но в этой книге, так же как в реальной жизни, мы можем иногда вольно использовать термин STL, подразумевая всю стандартную библиотеку, и наоборот.

Рассмотрим еще пару своих обобщенных алгоритмов, прежде чем углубиться в стандартные обобщенные алгоритмы, реализованные в STL. Вот шаблонная функция `count`, возвращающая общее количество элементов в контейнере:

```
template<class Container>
int count(const Container& container)
{
  int sum = 0;
  for (auto&& elt : container) {
    sum += 1;
  }
  return sum;
}
```

А вот функция `count_if`, возвращающая количество элементов, удовлетворяющих пользовательской функции-предикату:

```
template<class Container, class Predicate>
int count_if(const Container& container, Predicate pred)
{
  int sum = 0;
  for (auto&& elt : container) {
    if (pred(elt)) {
      sum += 1;
    }
  }
  return sum;
}
```

Эти функции можно использовать, как показано ниже:

```
std::vector<int> v = {3, 1, 4, 1, 5, 9, 2, 6};

assert(count(v) == 8);

int number_above =
  count_if(v, [](int e) { return e > 5; });
int number_below =
  count_if(v, [](int e) { return e < 5; });

assert(number_above == 2);
assert(number_below == 5);
```

Как много заключено в этом маленьком выражении `pred(elt)`! Попробуйте реализовать функцию `count_if` в терминах классического полиморфизма, просто чтобы понять, насколько быстро все начнет разваливаться. Под синтаксическим сахаром современного C++ скрыто огромное количество сигнатур. Например, синтаксис цикла `for` с диапазонами в нашей функции `count_if` преобразуется (или упрощается) компилятором в цикл `for`, действующий в терминах методов `container.begin()` и `container.end()`, каждый из которых должен возвращать итератор с типом, зависящим от типа контейнера. Другой пример: в обобщенной версии мы нигде не указываем – и нам нигде не нужно указывать, – будет ли `pred` принимать свой параметр `elt` по значению или по ссылке. Попробуйте реализовать то же самое в `virtual bool operator()`!

В продолжение темы итераторов: возможно, вы заметили, что все наши функции в этой главе (мономорфные, полиморфные или обобщенные) выражены в терминах контейнеров. В своей версии `count` мы подсчитывали элементы во всем контейнере. В `count_if` мы подсчитывали элементы во всем контейнере, соответствующие заданному условию. Как оказывается, это очень распространенный способ записи алгоритмов, особенно в современном C++; поэтому многое из того, что мы ожидаем увидеть в алгоритмах работы с контейнерами (или в их близких родственниках – алгоритмах работы с диапазонами), перекочует в C++20 или C++23. Однако STL зародилась в далеких 1990-х, до появления современного C++. Поэтому авторы STL предполагали, что обработка контейнеров будет обходиться особенно дорого (из-за всех этих дорогостоящих вызовов конструкторов копий – напомню, что семантика перемещения и конструкторы перемещения появились только в C++11) и проектировали STL в основном для работы с легковесной концепцией – итераторами. Это станет темой нашей следующей главы.

Итоги

Классический полиморфизм и обобщенное программирование призваны решить проблему параметризации поведения алгоритма: например, чтобы дать возможность написать функцию поиска, которая способна работать с произвольными операциями сопоставления.

Классический полиморфизм решает проблему за счет определения *базового класса* с закрытым списком *виртуальных функций-членов*, и реализации *полиморфных функций*, принимающих указатели или ссылки на экземпляры конкретных классов, *наследующих* базовый класс.

Обобщенное программирование решает ту же проблему за счет определения *концепции* с закрытым набором *требований* и создания экземпляров *шаблонных функций* с конкретными классами, *моделирующими* эту концепцию.

Классический полиморфизм испытывает проблемы с параметризацией высокого уровня (например, манипулирование объектами функций с любой сигнатурой) и с отношениями между типами (например, манипулирование элементами произвольных контейнеров). Поэтому в STL большое внимание уделяется обобщенным реализациям на основе шаблонов и почти полностью отсутствует классический полиморфизм.

Применение приемов обобщенного программирования помогает справиться с проблемами, если учитываются концептуальные требования к типам; но компилятор C++, по крайней мере соответствующий стандарту C++17, пока не в состоянии напрямую помочь с проверкой этих требований.

Глава 2

Итераторы и диапазоны

В предыдущей главе мы реализовали несколько обобщенных алгоритмов, оперирующих контейнерами, но избранный нами подход был очень неэффективен.

В этой главе вы узнаете:

- как и почему C++ обобщает идею указателей для создания концепции *итераторов*;
- важность *диапазонов* в C++ и стандартный способ выражения полуоткрытого диапазона как пары итераторов;
- как реализовать собственные надежные константные типы итераторов;
- как писать обобщенные алгоритмы, оперирующие парами итераторов;
- стандартная иерархия итераторов и их алгоритмическая важность.

Проблема целочисленных индексов

В предыдущей главе мы реализовали несколько обобщенных алгоритмов, оперирующих контейнерами. Взглянем на них еще раз:

```
template<typename Container>
void double_each_element(Container& arr)
{
  for (int i=0; i < arr.size(); ++i) {
    arr.at(i) *= 2;
  }
}
```

Этот алгоритм определен в терминах низкоуровневых операций `.size()` и `.at()`. В целом такое решение хорошо работает с такими контейнерными типами, как `array_of_ints` или `std::vector`, но намного хуже, например, со связными списками, такими как `list_of_ints` из предыдущей главы:

```
class list_of_ints {
  struct node {
    int data;
    node *next;
  };
```

```
    node *head_ = nullptr;
    node *tail_ = nullptr;
    int size_ = 0;
  public:
    int size() const { return size_; }
    int& at(int i) {
      if (i >= size_) throw std::out_of_range("at");
      node *p = head_;
      for (int j=0; j < i; ++j) {
        p = p->next;
      }
      return p->data;
    }

    void push_back(int value) {
      node *new_tail = new node{value, nullptr};
      if (tail_) {
        tail_->next = new_tail;
      } else {
        head_ = new_tail;
      }
      tail_ = new_tail;
      size_ += 1;
    }

    ~list_of_ints() {
      for (node *next, *p = head_; p != nullptr; p = next) {
        next = p->next;
        delete p;
      }
    }
  };
```

Реализация `list_of_ints::at()` имеет сложность O(n), зависящую от длины списка: чем длиннее список, тем медленнее работает `at()`. А когда наша функция `count_if` выполняет обход всех элементов списка, она n раз вызывает функцию `at()`, увеличивая сложность обобщенного алгоритма до O(n^2), – и это для простой операции подсчета, сложность которой должна быть O(n)!

Как оказывается, целочисленное индексирование с `.at()` является не лучшим фундаментом для возведения алгоритмических зáмков. Мы должны выбрать простую операцию, которая ближе к тому, как компьютеры обрабатывают данные на самом деле.

За границами указателей

В отсутствие любых абстракций как обычно идентифицируются элементы массива, связного списка или дерева? Самый простой способ – использовать указатель с адресом элемента в памяти. На рис. 2.1 показано несколько примеров указателей на элементы разных структур данных.

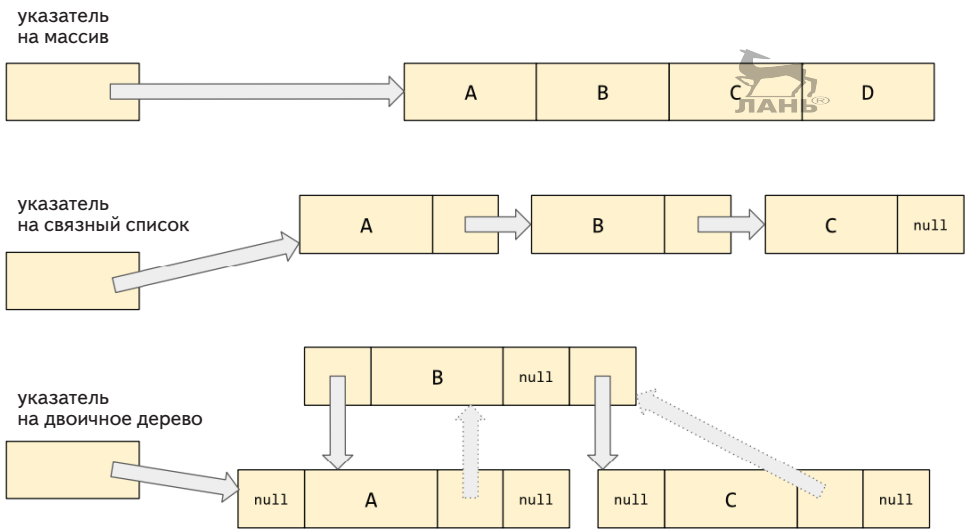


Рис. 2.1. Указатели на элементы разных структур данных

Чтобы выполнить обход элементов *массива*, нам достаточно простого указателя; мы можем обработать все элементы массива, начав с указателя на первый элемент и потом просто наращивая указатель, пока не будет достигнут последний элемент. В языке C:

```
for (node *p = lst.head_; p != nullptr; p = p->next) {
  if (pred(p->data)) {
    sum += 1;
  }
}
```

Но для эффективного обхода элементов *связного списка* простого указателя недостаточно; в результате инкремента указателя типа `node*` едва ли получится указатель на следующий узел в списке! В этом случае нужно нечто, действующее как указатель, – в частности, нужна возможность разыменовать это нечто, чтобы получить или изменить элемент списка, – но имеющее особое поведение, характерное для контейнера, связанное с абстрактной концепцией инкрементирования.

В C++, учитывая встроенную поддержку перегрузки операторов, когда я говорю: «особое поведение, связанное с абстрактной концепцией инкрементирования», – вы должны подумать: «перегружающее оператор ++». Так мы и поступим:

```
struct list_node {
  int data;
  list_node *next;
```

```
};

class list_of_ints_iterator {
  list_node *ptr_;

  friend class list_of_ints;
  explicit list_of_ints_iterator(list_node *p) : ptr_(p) {}
public:
  int& operator*() const { return ptr_->data; }
  list_of_ints_iterator& operator++()
    { ptr_ = ptr_->next; return *this; }
  list_of_ints_iterator operator++(int)
    { auto it = *this; ++*this; return it; }
  bool operator==(const list_of_ints_iterator& rhs) const
    { return ptr_ == rhs.ptr_; }
  bool operator!=(const list_of_ints_iterator& rhs) const
    { return ptr_ != rhs.ptr_; }
};

class list_of_ints {
  list_node *head_ = nullptr;
  list_node *tail_ = nullptr;
  // ...
public:
  using iterator = list_of_ints_iterator;
  iterator begin() { return iterator{head_}; }
  iterator end() { return iterator{nullptr}; }
};

template<class Container, class Predicate>
int count_if(Container& ctr, Predicate pred)
{
  int sum = 0;
  for (auto it = ctr.begin(); it != ctr.end(); ++it) {
    if (pred(*it)) {
      sum += 1;
    }
  }
  return sum;
}
```

Обратите внимание, что мы также перегрузили унарный оператор * (для разыменования) и операторы == и !=; наша шаблонная реализация `count_if` требует, чтобы все эти операции можно было использовать для управления переменной цикла `it`. (Технически наша функция `count_if` не требует операции ==, но если вы решили перегрузить один оператор сравнения, вы должны перегрузить и второй.)

Константные итераторы

Есть еще одно осложнение, которое нужно рассмотреть, прежде чем отказаться от этого примера итератора списка. Обратите внимание, что я втихомолку изменил нашу шаблонную функцию `count_if`, и теперь она принимает `Container&` вместо `const Container&`! Это потребовалось потому, что функции-члены `begin()` и `end()` являются неконстантными; а это, в свою очередь, объясняется тем, что они возвращают итераторы, оператор `operator*` которых возвращает неконстантные ссылки на элементы списка. Нам бы хотелось, чтобы наш тип списка (и его итераторы) имел полноценную константную корректность (const-correct) – то есть чтобы была возможность определять и использовать переменные типа `const list_of_ints` и предотвращать возможность изменять элементы константного списка.

В стандартной библиотеке эта проблема решается реализацией для каждого стандартного контейнера двух видов итераторов: `bag::iterator` и `bag::const_iterator`. Неконстантная функция-член `bag::begin()` возвращает `iterator`, а функция-член `bag::begin() const` возвращает `const_iterator`. Символ подчеркивания важен здесь! Обратите внимание, что `bag::begin() const` не возвращает простой `const iterator`; если бы возвращаемый ею итератор был константным, мы не смогли бы применить к нему оператор ++. (Что, в свою очередь, затруднило бы итерации по элементам `const bag`!) В действительности `bag::begin() const` возвращает нечто более затейливое: неконстантный объект `const_iterator`, оператор `operator*` которого просто возвращает *константную* ссылку на его элемент.

Возможно, пример поможет лучше понять эту идею. Давайте реализуем `const_iterator` для нашего контейнера `list_of_ints`.

Поскольку большая часть кода для типа `const_iterator` будет совпадать с кодом для типа `iterator`, первым инстинктивным желанием может стать стремление скопировать одинаковые блоки кода. Но это же C++! Когда я говорю «большая часть кода для одного типа будет совпадать с кодом для другого типа», вы должны сразу смекнуть: «а давайте заключим общие части в шаблон». Вот как мы поступим на самом деле:

```
struct list_node {
  int data;
  list_node *next;
};

template<bool Const>
class list_of_ints_iterator {
  friend class list_of_ints;
  friend class list_of_ints_iterator<!Const>;

  using node_pointer = std::conditional_t<Const, const list_node*, list_node*>;
  using reference = std::conditional_t<Const, const int&, int&>;
```

```
  node_pointer ptr_;

  explicit list_of_ints_iterator(node_pointer p) : ptr_(p) {}
public:
  reference operator*() const { return ptr_->data; }
  auto& operator++() { ptr_ = ptr_->next; return *this; }
  auto operator++(int) { auto result = *this; ++*this; return result; }

  // Поддержка сравнения типов iterator и const_iterator
  template<bool R>
  bool operator==(const list_of_ints_iterator<R>& rhs) const
    { return ptr_ == rhs.ptr_; }

  template<bool R>
  bool operator!=(const list_of_ints_iterator<R>& rhs) const
    { return ptr_ != rhs.ptr_; }

  // Поддержка неявного преобразования iterator в const_iterator
  // (но не наоборот)
  operator list_of_ints_iterator<true>() const
    { return list_of_ints_iterator<true>{ptr_}; }
};

class list_of_ints {
  list_node *head_ = nullptr;
  list_node *tail_ = nullptr;
  // ...
public:
  using const_iterator = list_of_ints_iterator<true>;
  using iterator = list_of_ints_iterator<false>;

  iterator begin() { return iterator{head_}; }
  iterator end() { return iterator{nullptr}; }
  const_iterator begin() const { return const_iterator{head_}; }
  const_iterator end() const { return const_iterator{nullptr}; }
};
```

Предыдущий код реализует полноценную константную корректность типов итераторов для нашего контейнера `list_of_ints`.

Пара итераторов определяет диапазон

Теперь, разобравшись с фундаментальным понятием итератора, применим его на практике. Мы уже знаем, что если есть пара итераторов, как те, что возвращаются функциями-членами `begin()` и `end()`, можно воспользоваться циклом `for` для обхода всех элементов, находящихся в контейнере. Но самое интересное, что пары итераторов можно использовать для обхода любого поддиапазона элементов! Допустим, мы решили просмотреть первую половину вектора:

```
template<class Iterator>
void double_each_element(Iterator begin, Iterator end)
{
  for (auto it = begin; it != end; ++it) {
    *it *= 2;
  }
}

int main()
{
  std::vector<int> v {1, 2, 3, 4, 5, 6};
  double_each_element(v.begin(), v.end());
    // удвоить все элементы в векторе
  double_each_element(v.begin(), v.begin()+3);
    // удвоить элементы в первой половине вектора
  double_each_element(&v[0], &v[3]);
    // еще раз удвоить элементы в первой половине вектора
}
```

Обратите внимание, что в первый и во второй вызовы `double_each_element` в функции `main()` мы передаем пару итераторов, порожденных из `v.begin()`; то есть два значения типа `std::vector::iterator`. В третий вызов мы передаем два значения типа `int*`. Так как в этом случае `int*` удовлетворяет всем требованиям типа итератора – а именно: поддерживает операции инкремента, сравнения и разыменования, – наш код прекрасно работает даже с указателями! Этот подход наглядно демонстрирует гибкость модели пар итераторов. (Но, вообще, желательно избегать простых указателей, если контейнер, такой как `std::vector`, предлагает полноценный тип `iterator`. Используйте итераторы, полученные из `begin()` и `end()`.)

Можно сказать, что пара итераторов неявно определяет диапазон элементов данных. И это верно для удивительно большого семейства алгоритмов! Нам не нужен доступ к контейнеру, чтобы выполнить некоторый поиск или преобразование; нам нужен доступ лишь к определенному диапазону элементов, где требуется произвести поиск или применить преобразование. Продолжая эту мысль, мы неизбежно приходим к идее *представления без владения* (non-owning view), которое относится к последовательности данных так же, как ссылка C++ к единственной переменной. Но представления и диапазоны – это более современные понятия, и прежде чем говорить о них, мы должны покончить с винтажной библиотекой STL образца 1998 года.

В предыдущем примере мы увидели первый настоящий обобщенный алгоритм в стиле STL. Вообще говоря, `double_each_element` не является особенно обобщенным алгоритмом в смысле реализации поведения, которое можно использовать в других программах; но эта версия функции теперь достаточно универсальна в смысле работы с парами итераторов, где итератор может быть любого типа, реализующего операции инкремента, сравнения и разыменования. (Еще более обобщенную версию этого алгоритма мы увидим в следующей главе, когда будем говорить о `std::transform`.)

Категории итераторов

А теперь вернемся к функциям count и count_if из главы 1 «Классический полиморфизм и обобщенное программирование». Сравните функции из главы 1 с определением шаблона в следующем примере; вы увидите, что они идентичны, за исключением подстановки пары итераторов (неявно определяющей диапазон) вместо параметра Container&, а также за исключением смены имени первой функции с count на distance. Я изменил имя потому, что эту функцию, почти в том же виде, как она определена здесь, можно найти в стандартной библиотеке шаблонов под именем std::distance и там же можно найти вторую функцию под именем std::count_if:

```
template<typename Iterator>
int distance(Iterator begin, Iterator end)
{
  int sum = 0;
  for (auto it = begin; it != end; ++it) {
    sum += 1;
  }
  return sum;
}

template<typename Iterator, typename Predicate>
int count_if(Iterator begin, Iterator end, Predicate pred)
{
  int sum = 0;
  for (auto it = begin; it != end; ++it) {
    if (pred(*it)) {
      sum += 1;
    }
  }
  return sum;
}

void test()
{
  std::vector<int> v = {3, 1, 4, 1, 5, 9, 2, 6};

  int number_above = count_if(v.begin(), v.end(), [](int e) { return e > 5; });
  int number_below = count_if(v.begin(), v.end(), [](int e) { return e < 5; });

  int total = distance(v.begin(), v.end()); // НЕОДНОЗНАЧНОЕ РЕШЕНИЕ

  assert(number_above == 2);
  assert(number_below == 5);
  assert(total == 8);
}
```

Рассмотрим строку в этом примере, отмеченную комментарием НЕОДНОЗНАЧНОЕ РЕШЕНИЕ. Она вычисляет расстояние между двумя итераторами, в

цикле наращивая первый, пока не будет достигнут второй. Насколько эффективным является это решение? Для некоторых видов итераторов – например, `list_of_ints::iterator` – у нас нет лучшего решения, чем это. Но для `vector::iterator` или `int*`, поддерживающего возможность итераций через непрерывные массивы данных, было бы глупо использовать цикл и алгоритм со сложностью O(*n*), когда тот же результат можно получить за время O(1) простым вычитанием указателей. То есть было бы неплохо иметь в стандартной библиотеке версию `std::distance` для включения шаблонной специализации, как показано ниже:

```
template<typename Iterator>
int distance(Iterator begin, Iterator end)
{
  int sum = 0;
  for (auto it = begin; it != end; ++it) {
    sum += 1;
  }
  return sum;
}

template<>
int distance(int *begin, int *end)
{
  return end - begin;
}
```

Но нам нужно, чтобы специализация поддерживалась не только для `int*` и `std::vector::iterator`. Нам нужно, чтобы стандартная библиотечная функция `std::distance` действовала эффективно с любыми типами итераторов, поддерживающими определенные операции. То есть мы начинаем понимать, что есть (по крайней мере) два разных вида итераторов: поддерживающие инкремент, сравнение и разыменование и поддерживающие инкремент, сравнение, разыменование, *а также вычитание*! Как оказывается, для любого типа итераторов, где имеет смысл операция `i = p - q`, имеет смысл и обратная операция `q = p + i`. Итераторы, поддерживающие вычитание и сложение, называют *итераторами с произвольным доступом* (random-access iterators).

То есть стандартная библиотечная функция `std::distance` должна быть эффективной и для итераторов с произвольным доступом, и для других видов итераторов. Чтобы упростить частичную специализацию в подобных шаблонах, стандартная библиотека вводит идею иерархии видов итераторов. Итераторы, такие как `int*`, которые поддерживают сложение и вычитание, известны как итераторы с произвольным доступом. Мы говорим, что они удовлетворяют концепции `RandomAccessIterator`.

Итераторы, менее гибкие, чем итераторы с произвольным доступом, могут не поддерживать сложение или вычитание произвольных расстояний, но, по меньшей мере, поддерживают операции инкремента и декремента в виде `++p` и `--p`. Итераторы этого вида называют двунаправленными: `BidirectionalIterator`.

Все итераторы с произвольным доступом одновременно являются двунаправленными итераторами, но обратное утверждение верно не всегда. В некотором смысле можно представить `RandomAccessIterator` как подкласс или подчиненную концепцию по отношению к `BidirectionalIterator`; и мы можем сказать, что `BidirectionalIterator` является *более свободной концепцией*, предъявляющей меньше требований, чем `RandomAccessIterator`.

Еще более свободная концепция – вид итераторов, не поддерживающих операцию декремента. Например, наш тип `list_of_ints::iterator` не поддерживает декремент, потому что наш связный список не имеет указателей на предыдущий элемент; мы можем двигаться только вперед, от первого элемента к последнему, и никогда – назад. Итераторы, поддерживающие `++p`, но не `--p`, называют *однонаправленными*, или *прямыми*, *итераторами* `ForwardIterator`. `ForwardIterator` – еще более свободная концепция, чем `BidirectionalIterator`.

Итераторы ввода и вывода

Мы можем вообразить еще более свободную концепцию, чем `ForwardIterator`! Например, одной из полезных операций с `ForwardIterator` является создание копии, ее сохранение и использование для повторного обхода тех же данных. Манипулирование итератором (или его копиями) никак не влияет на диапазон данных. Но мы могли бы изобрести итератор, как представленный в следующем фрагменте, где вообще нет данных, лежащих в основе, и даже не имеет смысла создавать копию итератора:

```
class getc_iterator {
  char ch;
public:
  getc_iterator() : ch(getc(stdin)) {}
  char operator*() const { return ch; }
  auto& operator++() { ch = getc(stdin); return *this; }
  auto operator++(int) { auto result(*this); ++*this; return result; }
  bool operator==(const getc_iterator&) const { return false; }
  bool operator!=(const getc_iterator&) const { return true; }
};
```

(Фактически стандартная библиотека содержит несколько типов итераторов, очень похожих на этот; мы обсудим один такой тип, `std::istream_iterator`, в главе 9 «Потоки ввода/вывода».) Такие итераторы, которые бессмысленно копировать и которые не указывают на элементы данных в сколько-нибудь значимом смысле, называют итераторами ввода `InputIterator`.

Возможен также зеркальный случай. Взгляните на следующий вымышленный тип итератора:

```
class putc_iterator {
  struct proxy {
```

```
    void operator= (char ch) { putc(ch, stdout); }
  };
public:
  proxy operator*() const { return proxy{}; }
  auto& operator++() { return *this; }
  auto& operator++(int) { return *this; }
  bool operator==(const putc_iterator&) const { return false; }
  bool operator!=(const putc_iterator&) const { return true; }
};

void test()
{
  putc_iterator it;
  for (char ch : {'h', 'e', 'l', 'l', 'o', '\n'}) {
    *it++ = ch;
  }
}
```

(И снова в стандартной библиотеке имеются типы итераторов, очень похожие на этот; например, `std::back_insert_iterator`, рассматриваемый в главе 3 «Алгоритмы с парами итераторов», и `std::ostream_iterator`, рассматриваемый в главе 9 «Потоки ввода/вывода».) Такие итераторы, которые бессмысленно копировать и которые предоставляют доступ на запись, но не на чтение, называют итераторами вывода `OutputIterator`.

Любой тип итератора в C++ попадает, по меньшей мере, в одну из следующих категорий:

- `InputIterator`;
- `OutputIterator`;
- `ForwardIterator`;
- `BidirectionalIterator`;
- `RandomAccessIterator`.

Обратите внимание, что во время компиляции легко определить соответствие данного типа итератора требованиям `BidirectionalIterator` или `RandomAccessIterator`, но нет никакой возможности понять (исходя только из поддерживаемых синтаксических операций), имеем ли мы дело с `InputIterator`, `OutputIterator` или `ForwardIterator`. Рассмотренные нами примеры итераторов `getc_iterator`, `putc_iterator` и `list_of_ints::iterator` поддерживают один и тот же синтаксический набор операций: разыменование `*it`, инкремент `++it` и сравнение `it != it`. Эти три класса отличаются только на семантическом уровне. Но как стандартная библиотека различает их?

Как оказывается, стандартная библиотека нуждается в некоторой помощи со стороны разработчика нового типа итераторов. Алгоритмы в стандартной библиотеке работают только с классами итераторов, которые определяют *член* `typedef` с именем `iterator_category`. То есть:

```
class getc_iterator {
  char ch;
public:
  using iterator_category = std::input_iterator_tag;

  // ...
};

class putc_iterator {
  struct proxy {
    void operator= (char ch) { putc(ch, stdout); }
  };
public:
  using iterator_category = std::output_iterator_tag;

  // ...
};

template<bool Const>
class list_of_ints_iterator {
  using node_pointer = std::conditional_t<Const, const list_node*, list_node*>;
  node_pointer ptr_;

public:
  using iterator_category = std::forward_iterator_tag;

  // ...
};
```

В этом случае любой стандартный (или даже нестандартный) алгоритм, настраивающий свое поведение, опираясь на категории итераторов параметров типа своего шаблона, может делать это, просто исследуя `iterator_category`.

Категории итераторов, описанные выше, соответствуют следующим пяти стандартным тегам типов, объявленным в заголовке `<iterator>`:

```
struct input_iterator_tag { };
struct output_iterator_tag { };
struct forward_iterator_tag : public input_iterator_tag { };
struct bidirectional_iterator_tag : public forward_iterator_tag { };
struct random_access_iterator_tag : public bidirectional_iterator_tag { };
```

Обратите внимание, что `random_access_iterator_tag` фактически является производным (в классическом объектно-ориентированном смысле «полиморфизм-класс-иерархия») от `bidirectional_iterator_tag` и так далее: *концептуальная иерархия* видов итераторов отражается в *иерархию классов* `iterator_category`. Это может пригодиться в метапрограммировании шаблонов, когда требуется проверить теги; но для работы со стандартной библиотекой вам достаточно просто знать, что если вы когда-либо захотите передать `iterator_category` в функцию, тег типа `random_access_iterator_tag` будет

соответствовать функции, ожидающей аргумент типа `bidirectional_iterator_tag`:

```
void foo(std::bidirectional_iterator_tag t [[maybe_unused]])
{
  puts("std::vector's iterators are indeed bidirectional...");
}

void bar(std::random_access_iterator_tag)
{
  puts("...and random-access, too!");
}

void bar(std::forward_iterator_tag)
{
  puts("forward_iterator_tag is not as good a match");
}

void test()
{
  using It = std::vector<int>::iterator;
  foo(It::iterator_category{});
  bar(It::iterator_category{});
}
```


В этой точке у многих из вас возникнет вопрос: «А как же `int*`? Как определить член `typedef` для чего-то, что вообще является не классом, а простым скалярным типом? Скалярные типы не могут иметь членов `typedef`». Эту проблему, как и многие другие, возникающие в разработке программного обеспечения, можно решить, добавив дополнительный уровень косвенности. Вместо прямой ссылки на `T::iterator_category` стандартные алгоритмы всегда ссылаются на `std::iterator_traits<T>::iterator_category`. Шаблонный класс `std::iterator_traits<T>` соответственно специализируется для случая, когда `T` является типом указателя.

Кроме того, `std::iterator_traits<T>` оказался удобным местом для подвешивания других членов `typedef`. Он предоставляет следующие пять членов `typedef`, если и только если сам `T` предоставляет все пять (или если `T` является типом указателя): `iterator_category`, `difference_type`, `value_type`, `pointer` и `reference`.

Объединяем все вместе

Объединив знания, полученные в этой главе, мы теперь можем написать код, как в следующем примере. В нем представлена реализация нашего типа `list_of_ints` с нашим классом итератора (включая версию `const_iterator`), которая может использоваться стандартной библиотекой благодаря наличию всех пяти важных членов `typedef`.

```
struct list_node {
  int data;
  list_node *next;
};

template<bool Const>
class list_of_ints_iterator {
  friend class list_of_ints;
  friend class list_of_ints_iterator<!Const>;

  using node_pointer = std::conditional_t<Const, const list_node*, list_node*>;
  node_pointer ptr_;

  explicit list_of_ints_iterator(node_pointer p) : ptr_(p) {}
public:
  // Члены typedef для поддержки std::iterator_traits
  using difference_type = std::ptrdiff_t;
  using value_type = int;
  using pointer = std::conditional_t<Const, const int*, int*>;
  using reference = std::conditional_t<Const, const int&, int&>;
  using iterator_category = std::forward_iterator_tag;

  reference operator*() const { return ptr_->data; }
  auto& operator++() { ptr_ = ptr_->next; return *this; }
  auto operator++(int) { auto result = *this; ++*this; return result; }

  // Поддержка сравнения между типами iterator и const_iterator
  template<bool R>
  bool operator==(const list_of_ints_iterator<R>& rhs) const
    { return ptr_ == rhs.ptr_; }

  template<bool R>
  bool operator!=(const list_of_ints_iterator<R>& rhs) const
    { return ptr_ != rhs.ptr_; }

  // Поддержка неявного преобразования iterator в const_iterator
  // (но не наоборот)
  operator list_of_ints_iterator<true>() const
    { return list_of_ints_iterator<true>{ptr_}; }
};

class list_of_ints {
  list_node *head_ = nullptr;
  list_node *tail_ = nullptr;
  int size_ = 0;
public:
  using const_iterator = list_of_ints_iterator<true>;
  using iterator = list_of_ints_iterator<false>;

  // Функции-члены begin и end
  iterator begin() { return iterator{head_}; }
```

```
  iterator end() { return iterator{nullptr}; }
  const_iterator begin() const { return const_iterator{head_}; }
  const_iterator end() const { return const_iterator{nullptr}; }

  // Другие поддерживаемые операции
  int size() const { return size_; }
  void push_back(int value) {
    list_node *new_tail = new list_node{value, nullptr};
    if (tail_) {
      tail_->next = new_tail;
    } else {
      head_ = new_tail;
    }
    tail_ = new_tail;
    size_ += 1;
  }

  ~list_of_ints() {
    for (list_node *next, *p = head_; p != nullptr; p = next) {
      next = p->next;
      delete p;
    }
  }
};
```

Теперь, чтобы показать, что мы достаточно полно понимаем, как стандартная библиотека реализует обобщенные алгоритмы, напишем шаблонные функции `distance` и `count_if` так, как они могли бы быть реализованы в стандартной библиотеке C++17.

Обратите внимание, что в `distance` используется новый для C++17 синтаксис `if constexpr`. В этой книге мы не будем много рассуждать о новшествах, появившихся в базовом языке C++17, но в данном случае важно отметить, что синтаксис `if constexpr` можно использовать для устранения массы типового кода, который приходилось писать в C++14.

```
template<typename Iterator>
auto distance(Iterator begin, Iterator end)
{
  using Traits = std::iterator_traits<Iterator>;
  if constexpr (std::is_base_of_v<std::random_access_iterator_tag,
                typename Traits::iterator_category>) {
    return (end - begin);
  } else {
    auto result = typename Traits::difference_type{};
    for (auto it = begin; it != end; ++it) {
```

```
      ++result;
    }
    return result;
  }
}

template<typename Iterator, typename Predicate>
auto count_if(Iterator begin, Iterator end, Predicate pred)
{
  using Traits = std::iterator_traits<Iterator>;
  auto sum = typename Traits::difference_type{};
  for (auto it = begin; it != end; ++it) {
    if (pred(*it)) {
      ++sum;
    }
  }
  return sum;
}

void test()
{
  list_of_ints lst;
  lst.push_back(1);
  lst.push_back(2);
  lst.push_back(3);
  int s = count_if(lst.begin(), lst.end(), [](int i){
    return i >= 2;
  });
  assert(s == 2);
  int d = distance(lst.begin(), lst.end());
  assert(d == 3);
}
```

В следующей главе мы закончим практику реализации с нуля такого большого количества собственных шаблонных функций и займемся исследованием шаблонных функций из библиотеки STL. Но, прежде чем оставить это глубокое обсуждение итераторов, рассмотрим еще кое-что, о чем я хотел бы поговорить.

Устаревший std::iterator

У многих из вас наверняка возникла мысль: «В реализации каждого класса итератора я должен приводить те же самые пять членов `typedef`. Но это довольно большой объем типового кода, нельзя ли избежать его ввода?» Существует ли средство, позволяющее устранить весь этот типовой код?

Начиная со стандарта C++98 и вплоть до C++17 стандартная библиотека включала вспомогательный шаблонный класс, специально предназначенный для этого. Он назывался `std::iterator` и требовал указать пять параметров шаблонных типов, соответствующих пяти членам `typedef`, необходимых для `std::iterator_traits`. Три из этих параметров имели «разумные значения

по умолчанию», то есть простейший вариант его использования выглядел достаточно просто:

```
namespace std {
  template<
    class Category,
    class T,
    class Distance = std::ptrdiff_t,
    class Pointer = T*,
    class Reference = T&
  > struct iterator {
    using iterator_category = Category;
    using value_type = T;
    using difference_type = Distance;
    using pointer = Pointer;
    using reference = Reference;
  };
}

class list_of_ints_iterator :
public std::iterator<std::forward_iterator_tag, int>
{
  // ...
};
```


К сожалению для `std::iterator`, реальность оказалась намного сложнее, и по некоторым причинам, которые мы обсудим далее, в C++17 класс `std::iterator` был объявлен устаревшим.

Как мы видели в разделе «Константные итераторы», для поддержки константной корректности мы должны реализовать тип константного итератора вдобавок к каждому типу «неконстантного итератора». В итоге получается код, как в следующем примере:

```
template<
  bool Const,
  class Base = std::iterator<
    std::forward_iterator_tag,
    int,
    std::ptrdiff_t,
    std::conditional_t<Const, const int*, int*>,
    std::conditional_t<Const, const int&, int&>
  >
>
class list_of_ints_iterator : public Base
{
  using typename Base::reference; // Жутко неуклюже!

  using node_pointer = std::conditional_t<Const, const list_node*, list_node*>;
  node_pointer ptr_;

public:
```

```
    reference operator*() const { return ptr_->data; }
    // ...
};
```

Предыдущий код тяжелее читать или писать, чем версию без использования `std::iterator`; а кроме того, использование `std::iterator` предназначенным способом усложняет код с *открытым наследованием*, то есть близко напоминающий классическую объектно-ориентированную иерархию классов. У начинающих программистов вполне может возникнуть соблазн использовать такую иерархию классов при написании, например, таких функций:

```
template<typename... Ts, typename Predicate>
int count_if(const std::iterator<Ts...>& begin,
             const std::iterator<Ts...>& end,
             Predicate pred);
```

Внешне это похоже на наши примеры «полиморфного программирования» из главы 1 «Классический полиморфизм и обобщенное программирование», функции, реализующей разное поведение, принимая параметры с типом ссылки на базовый класс. Но в случае с `std::iterator` это сходство чисто случайно и вводит в заблуждение; наследование `std::iterator` не дает полиморфной иерархии классов, и ссылка на этот «базовый класс» из наших собственных функций является ошибкой!

Поэтому стандарт C++17 объявил устаревшим класс `std::iterator`, с прицелом на полное его удаление в стандарте 2020 года или более позднем. Вы не должны использовать `std::iterator` в своем коде.

Однако если в своем проекте вы используете библиотеку Boost, обратите внимание на ее эквивалент `std::iterator` с именем `boost::iterator_facade`. В отличие от `std::iterator`, базовый класс `boost::iterator_facade` предоставляет реализацию для раздражающих функций-членов, таких как `operator++(int)` и `operator!=`, которые в иных случаях превращаются в досадный типовой код. Чтобы задействовать `iterator_facade`, просто унаследуйте его и определите несколько простейших функций-членов, выполняющих операции разыменования, инкремента и определения равенства. (Так как наш итератор для списка является прямым итератором `ForwardIterator`, это все, что требуется. Для двунаправленного итератора `BidirectionalIterator` также нужно добавить реализацию функции-члена декремента и т. д.)

Поскольку эти функции-члены являются приватными, мы можем открыть доступ к ним для Boost, добавив объявление `friend class boost::iterator_core_access;`:

```
#include <boost/iterator/iterator_facade.hpp>

template<bool Const>
class list_of_ints_iterator : public boost::iterator_facade<
  list_of_ints_iterator<Const>,
  std::conditional_t<Const, const int, int>,
```

```
    std::forward_iterator_tag
  >
  {
    friend class boost::iterator_core_access;
    friend class list_of_ints;
    friend class list_of_ints_iterator<!Const>;

    using node_pointer = std::conditional_t<Const, const list_node*, list_node*>;
    node_pointer ptr_;

    explicit list_of_ints_iterator(node_pointer p) : ptr_(p) {}

    auto& dereference() const { return ptr_->data; }
    void increment() { ptr_ = ptr_->next; }

    // Поддержка сравнения между типами iterator и const_iterator
    template<bool R>
    bool equal(const list_of_ints_iterator<R>& rhs) const {
      return ptr_ == rhs.ptr_;}

  public:
    // Поддержка неявного преобразования iterator в const_iterator,
    // (но не наоборот)
    operator list_of_ints_iterator<true>() const
      { return list_of_ints_iterator<true>{ptr_}; }
  };
```

Обратите внимание, что первый аргумент шаблонного типа в `boost::iterator_facade` всегда является классом, определение которого вы пишете: это шаблон проектирования «Странно рекурсивный шаблон» (Curiously Recurring Template Pattern), с которым мы снова встретимся в главе 6 «Умные указатели».

Эта реализация итератора для списка с использованием `boost::iterator_facade` получилась намного короче аналогичной реализации из предыдущего раздела; экономия в основном обусловлена отсутствием повторяющейся реализации операторов отношения. Так как наш итератор для списка является `ForwardIterator`, мы должны реализовать только два оператора отношения; но если бы мы писали `RandomAccessIterator`, тогда `iterator_facade` мог бы сгенерировать реализации по умолчанию для операторов –, <, >, <= и >=, основанные на единственной простейшей функции-члене `distance_to`.

Итоги

В этой главе мы узнали, что обход элементов является одной из фундаментальных операций, которые обычно реализуются в структурах данных. Однако простых указателей часто недостаточно для обхода сложных структур: применение ++ к простому указателю нередко не «приводит к следующему элементу».

Библиотека Standard Template Library в C++ предлагает идею итератора как обобщенное представление указателей. Два итератора определяют *диапазон* данных. Этот диапазон может быть лишь частью содержимого контейнера или просто манипулировать определенной областью памяти, как было показано на примере `getc_iterator` и `putc_iterator`. Некоторые свойства типа итератора определяются его принадлежностью к категории итераторов – ввода, вывода, прямые, двунаправленные или с произвольным доступом – и могут использоваться шаблонными функциями для выбора быстрейшего из алгоритмов, доступных для определенной категории итераторов.

Определяя свой контейнерный тип, вы должны также определить свои типы итераторов обеих версий: константные и неконстантные. Шаблоны открывают удобную возможность для этого. Реализуя свои типы итераторов, избегайте устаревшего класса `std::iterator` и подумайте об использовании `boost::iterator_facade`.

Глава 3

Алгоритмы с парами итераторов

Теперь, когда вы познакомились с типами итераторов – стандартными и пользовательскими, – самое время посмотреть, что можно делать с их помощью.

В этой главе вы узнаете:

- что такое «полуоткрытый диапазон» – понятие, которое точно определяет *диапазон*, образуемый двумя итераторами;
- как определить категорию каждого стандартного алгоритма из числа: «только для чтения», «только для записи», «преобразующий» или «перестановочный»; а также из числа: «однодиапазонный», «двухдиапазонный» и «полуторадиапазонный»;
- что некоторые стандартные алгоритмы, такие как `merge` и `make_heap`, являются единственными строительными блоками, необходимыми для создания высокоуровневых сущностей, таких как `stable_sort` и `priority_queue`;
- как отсортировать диапазон, опираясь на компаратор, отличный от `operator<`;
- как управлять сортированными массивами с использованием *идиомы erase-remove*.

Замечание о заголовках

Большинство шаблонных функций, обсуждаемых в этой главе, объявлено в стандартном заголовке `<algorithm>`. Специальные типы итераторов, напротив, обычно объявляются в заголовке `<iterator>`. Если у вас возникнет вопрос, где искать конкретную сущность, я рекомендую обратиться за ответом к онлайн-справочнику, такому как cppreference.com; самостоятельно найти ответ порой очень непросто!

Диапазонные алгоритмы только для чтения

В предыдущей главе мы реализовали два алгоритма: `distance` и `count_if`. Оба они присутствуют в стандартной библиотеке.

`std::count_if(a,b,p)` возвращает число элементов между `a` и `b`, удовлетворяющих функции-предикату `p`, – то есть число элементов `e`, для которых вызов `p(e)` возвращает `true`.

Обратите внимание, что каждый раз, говоря «между a и b», мы имеем в виду диапазон, включающий *a, но не включающий *b, – в математике это называется «полуоткрытым диапазоном» и записывается с использованием асимметричной нотации [a,b). Почему мы не должны включать *b? Дело вот в чем: если b соответствует результату end() некоторого вектора, тогда он не должен указывать ни на какой элемент этого вектора! То есть в общем случае разыменование *конечной точки* диапазона является опасной операцией. Другая причина состоит в том, что полуоткрытые диапазоны удобно использовать для представления диапазонов с нулевой длиной; например, диапазон «от x до x» – это диапазон с нулевой длиной, не содержащий элементов данных.

Полуоткрытые диапазоны вполне естественны для C++, так же как в C. На протяжении десятилетий мы писали циклы for, выполняющие обход диапазонов от нижней границы (включительно) до верхней (не включая ее); эта идиома настолько распространена, что любое отклонение от нее часто указывает на ошибку:

```
constexpr int N = 10;
int a[N];

// Правильный цикл for.
for (int i=0; i < N; ++i) {
  // ...
}

// Один из вариантов "подозрительного" цикла for.
for (int i=0; i <= N; ++i) {
  // ...
}

// Правильный вызов стандартного алгоритма.
std::count_if(std::begin(a), std::end(a), [](int){ return true; });

// "Подозрительный" вызов.
std::count_if(std::begin(a), std::end(a) - 1, [](int){ return true; });

// "Тривиальный" вызов: подсчитывает элементы в диапазоне с нулевой длиной.
std::count_if(std::begin(a), std::begin(a), [](int){ return true; });
```

std::distance(a,b) возвращает количество элементов между a и b – то есть сколько раз потребуется применить операцию ++, чтобы достигнуть b. Можно считать, что эта функция действует эквивалентно вызову std::count_if(a,b,[](auto&&){return true;}).

Как было показано в главе 2 «Итераторы и диапазоны», если данный итератор является итератором с произвольным доступом, это количество можно быстро определить как (b - a), и стандартная std::distance именно так и поступает. Обратите внимание, что операция (b - a) может вернуть отрицательное число, если передать аргументы в «неправильном» порядке!

```
int a[] {1, 2, 3, 4, 5};
std::list<int> lst {1, 2, 3, 4, 5};
std::forward_list<int> flst {1, 2, 3, 4, 5};

assert(std::distance(std::begin(a), std::end(a)) == 5);
assert(std::distance(std::begin(flst), std::end(flst)) == 5);
assert(std::distance(std::begin(flst), std::end(flst)) == 5);

assert(std::distance(std::end(a), std::begin(a)) == -5);
```

Когда `std::distance` получает итераторы с произвольным доступом, она просто выполняет вычитание; поэтому можно сказать, что передача аргументов в «неправильном порядке» явно поддерживается и одобрена стандартом C++. Но если итераторы являются лишь двунаправленными (такими как `std::list<int>::iterator` – см. главу 4 «Зоопарк контейнеров»), «неверный порядок» не поддерживается. Вы могли бы ожидать, что сравнение `std::distance(b,a) == -std::distance(a,b)` должно возвращать `true` для итераторов всех типов; но подумайте, как сам алгоритм `std::distance` смог бы отличить «правильный порядок» следования итераторов от «неправильного»? Единственное, что он может сделать (в отсутствие `operator-`), – продолжать увеличивать `a` – возможно, уже за концом контейнера – и выйти в космос в тщетной надежде когда-нибудь достичь `b`:

```
// Следующая строка дает "неправильный" ответ!
// assert(std::distance(std::end(lst), std::begin(lst)) == 1);
// А эта просто вызывает ошибку сегментации!
// std::distance(std::end(flst), std::begin(flst));
```

Чтобы понять причины неправильного поведения кода в этом примере, обращайтесь к диаграммам `std::list` и `std::forward_list` в главе 4 «Зоопарк контейнеров».

`std::count(a,b,v)` возвращает количество элементов между `a` и `b`, равных `v`, то есть количество элементов `e`, для которых выполняется условие `e == v`. Можно считать, что эта функция действует эквивалентно вызову `std::count_if(a,b,[&v](auto&& e){return e == v;})`, и в действительности обе версии должны давать одинаковый код на ассемблере. Если бы в 1998 в C++ поддерживались лямбда-выражения, алгоритм `std::count` почти наверняка не попал бы в стандартную библиотеку.

Обратите внимание, что `std::count(a,b,v)` всегда просматривает все элементы в диапазоне между `a` и `b`. Он не может использовать специальную информацию о размещении данных в диапазоне, которая может иметься у вас. Например, представьте, что нам нужно подсчитать количество значений `42` в `std::set<int>`. Это можно реализовать любым из следующих способов:

```
std::set<int> s { 1, 2, 3, 10, 42, 99 };
bool present;

// O(n): сравнение каждого элемента с числом 42
present = std::count(s.begin(), s.end(), 42);

// O(log n): попросить контейнер самостоятельно отыскать число 42
present = s.count(42);
```

Алгоритм `std::count` проигрывает второму подходу, где запрос передается непосредственно самому множеству `set`. Контейнер преобразует обход всего множества со сложностью O(*n*) в поиск по дереву со сложностью O(log *n*). Аналогично `std::unordered_set` поддерживает метод `count`, имеющий примерную сложность O(1).

Дополнительную информацию об этих контейнерах вы найдете в главе 4 «Зоопарк контейнеров»; главное, чему учит этот пример, – иногда в наших данных имеется важная структура, которую можно использовать, выбирая правильный инструмент для решения задачи. Несмотря на то что я демонстрирую случаи, когда стандартные алгоритмы кажутся «волшебными», поступая правильно (как в случае алгоритма `std::distance`, выполняющего операцию `(b - a)`), вы не должны думать, что это «волшебство» простирается дальше, чем есть на самом деле. Стандартные алгоритмы знают ровно столько, сколько им сообщается, то есть им известны лишь свойства типов передаваемых итераторов. Они никогда не изменят свое поведение, опираясь на отношения элементов данных между собой. Организация проверки отношений между элементами данных (например, «эти данные отсортированы», «этот диапазон охватывает весь контейнер») – это ваша задача как программиста.

Вот еще несколько алгоритмов, похожих на `std::count` и `std::count_if`.

`std::find(a,b,v)` и `std::find_if(a,b,p)` работают точно так же, как `std::count(a,b,v)` и `std::count_if(a,b,p)` соответственно, с той лишь разницей, что вместо обхода всего диапазона и возврата *количества* элементов, соответствующих условию, варианты `find` перебирают элементы до первого совпадения и возвращают итератор, указывающий на найденный элемент данных. Существует также вариант `find_if_not`, подобный `find_if`, но он изменяет смысл предиката на обратный. Этого варианта тоже, возможно, не существовало бы, если бы в ранней истории C++ поддерживались лямбда-выражения:

```
template<class InputIterator, class UnaryPredicate>
InputIterator find_if(InputIterator first, InputIterator last, UnaryPredicate p)
{
  for (; first != last; ++first) {
    if (p(*first)) {
      return first;
    }
  }
  return last;
```

```
}

template<class It, class U>
It find_if_not(It first, It last, U p) {
  return std::find_if(first, last, [&](auto&& e){ return !p(e); });
}

template<class It, class T>
It find(It first, It last, T value) {
  return std::find_if(first, last, [&](auto&& e)
    { return e == value; });
}
```

Обратите внимание: так как алгоритм find возвращает управление сразу, как найдет первое совпадение, в среднем он работает быстрее, чем count (который просматривает весь диапазон, независимо от условия). Поведение такого вида, как «возвращает управление сразу», часто называют вычислением по короткой схеме.

std::all_of(a,b,p), std::any_of(a,b,p) и std::none_of(a,b,p) возвращают true или false, в зависимости от того, как часто функция-предикат p возвращает true для элементов в диапазоне. Все они могут быть реализованы на основе алгоритмов find, а значит, автоматически поддерживают вычисление по короткой схеме:

```
template<class It, class UnaryPredicate>
bool all_of(It first, It last, UnaryPredicate p)
{
  return std::find_if_not(first, last, p) == last;
}

template <class It, class U>
bool any_of(It first, It last, U p)
{
  return std::find_if(first, last, p) != last;
}

template <class It, class U>
bool none_of(It first, It last, U p)
{
  return std::find_if(first, last, p) == last;
}
```

Есть еще один find-подобный алгоритм: find_first_of. Он реализует операцию «поиск в последовательности первого вхождения любого целевого элемента из фиксированного множества» – то есть чем-то напоминает strcspn в стандартной библиотеке C, но может применяться к любым типам, не только к символам. Говоря отвлеченно, find_first_of принимает два концептуальных параметра: диапазон для поиска и множество целевых элементов. Так как это STL, они передаются на вход как диапазоны, то есть как пары итераторов. Со-

ответственно, вызов этого алгоритма выглядит как `find_first_of(haystack, haystack, needle, needle)`[1]: с двумя парами итераторов. Это может вызывать путаницу, поэтому будьте особенно внимательны с алгоритмами, принимающими несколько похожих параметров!

```
template <class It, class FwdIt>
It find_first_of(It first, It last, FwdIt targetfirst, FwdIt targetlast)
{
  return std::find_if(first, last, [&](auto&& e) {
    return std::any_of(targetfirst, targetlast, [&](auto&& t) {
      return e == t;
    });
  });
}

template <class It, class FwdIt, class BinaryPredicate>
It find_first_of(It first, It last, FwdIt targetfirst, FwdIt targetlast,
                 BinaryPredicate p)
{
  return std::find_if(first, last, [&](auto&& e) {
    return std::any_of(targetfirst, targetlast, [&](auto&& t) {
      return p(e, t);
    });
  });
}
```

Обратите внимание, что итераторы «haystack», как ожидается, могут быть любого старого типа `InputIterator`, но итераторы «needle» обязательно должны быть не менее чем `ForwardIterator`. Как рассказывалось в главе 2 «Итераторы и диапазоны», самое важное достоинство итераторов `ForwardIterator` – их можно копировать, чтобы получить возможность многократно обойти один и тот же диапазон. Именно это и требуется алгоритму `find_first_of`! Он обходит диапазон «needle» один раз для каждого символа в «haystack»; поэтому итераторы «needle» должны поддерживать возможность повторного обхода – и, кстати, определять диапазон конечного размера! Это требование конечности не относится к итераторам «haystack»; теоретически они могут возвращать элементы из бесконечного потока ввода:

```
std::istream_iterator<char> ii(std::cin);
std::istream_iterator<char> iend{};
std::string s = "hello";

// Продолжать извлекать символы из std::cin,
// пока не встретится 'h', 'e', 'l' или 'o'.
std::find_first_of(ii, iend, s.begin(), s.end());
```

Завершим обсуждение алгоритмов с несколькими схожими параметрами, рассмотрев простые алгоритмы `std::equal` и `std::mismatch`.

[1] Здесь игра слов: «haystack» переводится как «стог сена», а «needle» – иголка. – *Прим. перев.*

`std::equal(a,b,c,d)` принимает две пары итераторов: диапазон `[a,b)` и диапазон `[c,d)`. Возвращает `true`, если два диапазона поэлементно равны, и `false` в противном случае.

`std::mismatch(a,b,c,d)` похож на `find`: этот алгоритм точно сообщит, какая пара элементов вызвала несовпадение:

```
template<class T> constexpr bool is_random_access_iterator_v =
  std::is_base_of_v<std::random_access_iterator_tag, typename
  std::iterator_traits<T>::iterator_category>;

template<class It1, class It2, class B>
auto mismatch(It1 first1, It1 last1, It2 first2, It2 last2, B p)
{
  while (first1 != last1 && first2 != last2 && p(*first1, *first2)) {
    ++first1;
    ++first2;
  }
  return std::make_pair(first1, first2);
}

template<class It1, class It2>
auto mismatch(It1 first1, It1 last1, It2 first2, It2 last2)
{
  return std::mismatch(first1, last1, first2, last2, std::equal_to<>{});
}

template<class It1, class It2, class B>
bool equal(It1 first1, It1 last1, It2 first2, It2 last2, B p)
{
  if constexpr (is_random_access_iterator_v<It1> &&
                is_random_access_iterator_v<It2>) {
    // Диапазоны разной длины не могут быть равными.
    if ((last2 - first2) != (last1 - first1)) {
      return false;
    }
  }

  return std::mismatch(first1, last1, first2, last2, p) ==
    std::make_pair(last1, last2);
}

template<class It1, class It2>
bool equal(It1 first1, It1 last1, It2 first2, It2 last2)
{
  return std::equal(first1, last1, first2, last2, std::equal_to<>{});
}
```

Обратите внимание, что в качестве объекта предиката используется `std::equal_to<>{}`; мы не будем подробно рассматривать предикаты в этой книге, поэтому просто допустим, что `std::equal_to<>{}` – это объект, напо-

минающий поведением `[](auto a, auto b){ return a == b; }`, но с более *совершенной передачей* (perfect forwarding).

Наконец, будьте внимательны! Многие двухдиапазонные алгоритмы в стандартной библиотеке C++17 также имеют варианты, известные как алгоритмы с полуторным диапазоном. Например, в дополнение к `std::mismatch(a,b,c,d)` имеется версия `std::mismatch(a,b,c)`, где «конечная точка» определяется выражением `c + std::distance(a, b)`. Если такая вычисленная «конечная точка» окажется за границами контейнера, значит, вам не повезло!

Поскольку «удача» никогда не была хорошим ответом на технический вопрос, стандарт C++17 добавил безопасные двухдиапазонные варианты многих таких полуторадиапазонных алгоритмов, имевшихся в C++14.

Манипулирование данными с std::copy

Мы только что познакомились с первыми двухдиапазонными алгоритмами. В заголовке `<algorithm>` определено много таких двухдиапазонных и родственных им полуторадиапазонных алгоритмов. Какой из таких алгоритмов считается простейшим?

Естественным выглядит ответ: «копирующий все элементы из первого диапазона во второй». И действительно, в STL имеется такой алгоритм с именем `std::copy`:

```
template<class InIt, class OutIt>
OutIt copy(InIt first1, InIt last1, OutIt destination)
{
  while (first1 != last1) {
    *destination = *first1;
    ++first1;
    ++destination;
  }
  return destination;
}
```

Обратите внимание, что это полуторадиапазонный алгоритм. Фактически в стандартной библиотеке отсутствует двухдиапазонная версия `std::copy`; выполняя копирование, мы предварительно проверяем наличие достаточного объема в буфере `destination`, поэтому проверка «нахождения внутри буфера» внутри цикла не только избыточна, но и ухудшает эффективность.

Я уже слышу, как вы восклицаете: «Какой ужас! Ведь это та же дикая логика, которая была заключена в `strcpy`, `sprintf` и `gets`! Это же явное приглашение к атакам на переполнение буфера!» Вы действительно могли бы так воскликнуть в отношении `gets`, но функция `gets` была исключена из стандартной библиотеки C++17. Вы имели бы полное право сказать то же самое в отношении `sprintf`, но всякий, кому нужна эта функция, предпочтет использовать версию с проверкой диапазона `snprintf`, которая в этом контексте подобна «двухдиапазонному алгоритму». Однако в отношении `strcpy` я не согласен. Используя `gets`, невоз-

можно заранее определить правильный размер приемного буфера; используя sprintf, это возможно, хотя и сложно; но при использовании strcpy это до смешного просто: достаточно вызвать strlen для входного буфера, и ответ у вас в кармане. То же относится к std::copy: «количество входных элементов» и «количество элементов на выходе» точно соответствуют друг другу, поэтому создание буфера правильного размера не представляет никакой проблемы.

Обратите внимание, что параметр с именем destination – это *итератор вывода*. Это означает, что std::copy можно использовать не только для копирования непрерывных блоков памяти, но также для передачи данных в произвольную функцию «вывода». Например:

```
class putc_iterator : public boost::iterator_facade<
  putc_iterator, // T
  const putc_iterator, // value_type
  std::output_iterator_tag
  >
{
  friend class boost::iterator_core_access;

  auto& dereference() const { return *this; }
  void increment() {}

  bool equal(const putc_iterator&) const { return false; }
public:
  // Этот итератор является собственным прокси-объектом!
  void operator= (char ch) const { putc(ch, stdout); }
};

void test()
{
  std::string s = "hello";
  std::copy(s.begin(), s.end(), putc_iterator{});
}
```

Возможно, вам будет интересно сравнить эту версию нашего итератора putc_iterator с версией из главы 2 «Итераторы и диапазоны»; эта версия использует boost::iterator_facade, как рекомендовалось в конце главы 2, и использует распространенный трюк, возвращая *this вместо нового прокси-объекта.

Теперь можно воспользоваться гибкостью destination для решения проблемы переполнения буфера! Допустим, что мы записываем данные не в фиксированный массив, а в std::vector, изменяющий свой размер динамически (см. главу 4 «Зоопарк контейнеров»). В этом случае «запись элемента» соответствует «вталкиванию элемента в конец» вектора. То есть можно написать выходной итератор, очень похожий на putc_iterator, который вызывает push_back вместо putc, и тем самым мы можем защититься от проблемы переполнения. На самом деле стандартная библиотека уже определяет такой итератор в заголовке <iterator>:

```
namespace std {
  template<class Container>
  class back_insert_iterator {
    using CtrValueType = typename Container::value_type;
    Container *c;
  public:
    using iterator_category = output_iterator_tag;
    using difference_type = void;
    using value_type = void;
    using pointer = void;
    using reference = void;

    explicit back_insert_iterator(Container& ctr) : c(&ctr) {}

    auto& operator*() { return *this; }
    auto& operator++() { return *this; }
    auto& operator++(int) { return *this; }

    auto& operator= (const CtrValueType& v) {
      c->push_back(v);
      return *this;
    }

    auto& operator= (CtrValueType&& v) {
      c->push_back(std::move(v));
      return *this;
    }
  };

  template<class Container>
  auto back_inserter(Container& c)
  {
    return back_insert_iterator<Container>(c);
  }
}

void test()
{
  std::string s = "hello";
  std::vector<char> dest;
  std::copy(s.begin(), s.end(), std::back_inserter(dest));
  assert(dest.size() == 5);
}
```

Вызов `std::back_inserter(dest)` просто возвращает объект `back_insert_iterator`. В C++17 можно положиться на автоматическое определение типа шаблона в конструкторах и определить тело этой функции просто как `return std::back_insert_iterator(dest);` или вообще отказаться от нее и напрямую использовать `std::back_insert_iterator(dest)` – где в коде C++14 можно было «довольствоваться» `std::back_inserter(dest)`. Так зачем нам мог бы

понадобиться этот, казалось бы, избыточный код? Имя `back_inserter` было выбрано намеренно, чтобы его проще было запомнить, потому что предполагается, что эта функция будет использоваться очень часто. Несмотря на то что C++17 позволяет писать `std::pair` вместо `std::make_pair` и `std::tuple` вместо `std::make_tuple`, было бы глупо писать более «заковыристое» имя `std::back_insert_iterator` вместо `std::back_inserter`. Поэтому даже в C++17 лучше писать `std::back_inserter(dest)`.

Вариации на тему: std::move и std::move_iterator

Как нетрудно догадаться из названия или из предыдущей реализации, алгоритм `std::copy` копирует элементы из входного диапазона в выходной. Знатоки C++11 могли бы задаться вопросом: а что, если вместо *копирования* элементов использовать семантику перемещения, чтобы *переместить* их из входного диапазона в выходной?

В STL имеется два разных подхода к решению этой задачи. Первый, самый простой – использовать алгоритм `std::move` (определен в заголовке `<algorithm>`), имеющий следующее определение:

```
template<class InIt, class OutIt>
OutIt move(InIt first1, InIt last1, OutIt destination)
{
  while (first1 != last1) {
    *destination = std::move(*first1);
    ++first1;
    ++destination;
  }
  return destination;
}
```

Он почти в точности повторяет алгоритм `std::copy`, за исключением дополнительного вызова `std::move` для входного элемента (будьте внимательны – этот внутренний алгоритм `std::move`, с единственным аргументом, определен в заголовке `<utility>`, и он полностью отличается от внешнего `std::move` с тремя аргументами, который определяется в заголовке `<algorithm>`! Конечно печально, что два разных алгоритма имеют одинаковые имена. По иронии судьбы, в число немногих других функций в STL, страдающих той же проблемой, входит `std::remove`; см. раздел «Удаление из сортированного массива» в конце этой главы, а также главу 12 «Файловая система»).

Другой подход основан на использовании решения, которое мы видели выше с `back_inserter`. Вместо того чтобы отключать основной алгоритм, мы можем продолжать использовать алгоритм `std::copy`, но параметризовать его иначе. Допустим, мы передаем в него новый тип итератора, который (подобно `back_inserter`) обертывает исходный объект и изменяет его поведение.

В частности, нам нужен итератор ввода, реализация operator* которого возвращает *rvalue*. Мы можем сделать это!

```
template<class It>
class move_iterator {
  using OriginalRefType = typename std::iterator_traits<It>::reference;
  It iter;
public:
  using iterator_category = typename
    std::iterator_traits<It>::iterator_category;
  using difference_type = typename
    std::iterator_traits<It>::difference_type;
  using value_type = typename std::iterator_traits<It>::value_type;
  using pointer = It;
  using reference = std::conditional_t<
    std::is_reference_v<OriginalRefType>,
    std::remove_reference_t<OriginalRefType>&&,
    OriginalRefType
    >;

  move_iterator() = default;

  explicit move_iterator(It it) : iter(std::move(it)) {}

  // Разрешить конструирование или присваивание из любого move-итератора.
  // Эти шаблоны также могут играть роль конструктора копирования
  // и оператора присваивания соответственно.
  template<class U>
  move_iterator(const move_iterator<U>& m) : iter(m.base()) {}
  template<class U>
  auto& operator=(const move_iterator<U>& m)
    { iter = m.base(); return *this; }

  It base() const { return iter; }

  reference operator*() { return static_cast<reference>(*iter); }
  It operator->() { return iter; }
  decltype(auto) operator[](difference_type n) const
    { return *std::move(iter[n]); }

  auto& operator++() { ++iter; return *this; }
  auto& operator++(int)
    { auto result = *this; ++*this; return result; }
  auto& operator--() { --iter; return *this; }
  auto& operator--(int)
    { auto result = *this; --*this; return result; }

  auto& operator+=(difference_type n) const
    { iter += n; return *this; }
  auto& operator-=(difference_type n) const
    { iter -= n; return *this; }
```

```
};

// Я опустил определения операторов нечленов
// == != < <= > >= + - ; сможете реализовать их самостоятельно?

template<class InputIterator>
auto make_move_iterator(InputIterator& c)
{
  return move_iterator(c);
}
```

Извиняюсь за такой большой фрагмент кода, но уверяю, что вы спокойно можете не вникать в его тонкости. Для тех, кому это нравится, здесь имеется шаблонный конструктор из `move_iterator<U>`, который, так получилось, дублирует конструктор копирования (когда тип `U` совпадает с типом `It`); а также множество функций-членов (таких как `operator[]` и `operator--`), тела которых будут вызывать ошибку во время компиляции для многих допустимых типов `It` (например, если `It` – прямой итератор; см. главу 2 «Итераторы и диапазоны»), но это нормально, потому что их тела не будут создаваться компилятором, если только пользователь на самом деле не попытается вызвать эти функции на этапе компиляции (если пользователь попробует это сделать – `move_iterator<list_of_ints::iterator>`, тогда, конечно, этот код вызовет ошибку времени компиляции).

Обратите внимание, что по аналогии с `back_inserter` в STL имеется вспомогательная функция `make_move_iterator` для компиляторов, появившихся перед выходом стандарта C++17, которые не поддерживают автоматический вывод типа шаблона конструктора. Имя «вспомогательной» функции, как в случае с `make_pair` и `make_tuple`, выглядит менее удобоваримым, чем фактическое имя класса, поэтому я настоятельно советую использовать возможности C++17; зачем вводить лишние пять символов и создавать лишний экземпляр шаблонной функции, если она вам не нужна?

Теперь у нас есть два разных способа перемещения данных из одного контейнера или диапазона в другой: алгоритм `std::move` и класс-адаптер `std::move_iterator`. Вот примеры обеих идиом:

```
std::vector<std::string> input = {"hello", "world"};
std::vector<std::string> output(2);

// Первый подход: использовать алгоритм std::move
std::move(input.begin(), input.end(), output.begin());

// Второй подход: использовать move_iterator
std::copy(
  std::move_iterator(input.begin()),
  std::move_iterator(input.end()),
  output.begin()
);
```

Первый подход, с использованием `std::move`, более очевидный, если вам требуется просто переместить данные. Но зачем потребовалось включать в стандартную библиотеку этот «малопонятный» `move_iterator`? Чтобы ответить на этот вопрос, мы должны исследовать еще один алгоритм, принципиально связанный с `std::copy`.

Непростое копирование с std::transform

Возможно, вы заметили при обсуждении реализации `std::copy`, что два параметра типа `value_type` в итераторе не обязательно должны совпадать. Это фича, а не баг! Благодаря ей можно писать код, полагающийся на неявные преобразования, и он просто «Будет Делать То, Что Должен»:

```
std::vector<const char *> input = {"hello", "world"};
std::vector<std::string> output(2);

std::copy(input.begin(), input.end(), output.begin());

assert(output[0] == "hello");
assert(output[1] == "world");
```

Выглядит тривиально, верно? Но пригладитесь внимательнее! Глубоко внутри нашего экземпляра `std::copy` будет вызван неявный конструктор, преобразующий `const char *` (тип `*input.begin()`) в `std::string` (тип `*output.begin()`). То есть уже в который раз мы видим пример обобщенного кода, выполняющего на удивление сложные операции просто потому, что ему передаются итераторы определенных типов.

Но иногда во время копирования бывает нужно применить функцию сложного преобразования – еще более сложного, чем может выполнить неявное преобразование. Стандартная библиотека поможет вам в этом!

```
template<class InIt, class OutIt, class Unary>
OutIt transform(InIt first1, InIt last1, OutIt destination, Unary op)
{
  while (first1 != last1) {
    *destination = op(*first1);
    ++first1;
    ++destination;
  }
  return destination;
}

void test()
{
  std::vector<std::string> input = {"hello", "world"};
  std::vector<std::string> output(2);

  std::transform(
```

```
    input.begin(),
    input.end(),
    output.begin(),
    [](std::string s) {
      // Преобразование на месте также возможно!
      std::transform(s.begin(), s.end(), s.begin(), ::toupper);
      return s;
    }
  );
  assert(input[0] == "hello");
  assert(output[0] == "HELLO");
}
```

Иногда даже бывает нужно применить для преобразования функцию, принимающую *два* аргумента. И снова библиотека позаботится об этом:

```
template<class InIt1, class InIt2, class OutIt, class Binary>
OutIt transform(InIt1 first1, InIt1 last1, InIt2 first2,
                OutIt destination, Binary op)
{
  while (first1 != last1) {
    *destination = op(*first1, *first2);
    ++first1;
    ++first2;
    ++destination;
  }
  return destination;
}
```

Эту версию алгоритма `std::transform` юмористически можно описать как «одно- и двухдиапазонный с половиной» алгоритм!

(А как насчет функции с тремя аргументами? А с четырьмя? К сожалению, полноценной вариативной (с переменным числом аргументов) версии `std::transform` не существует; вариативные шаблоны не поддерживались в C++ до появления стандарта C++11. Можете попробовать сами реализовать вариативную версию и посмотреть, с какими проблемами столкнетесь – они не тривиальны, но вполне преодолимы.)

Существование `std::transform` дает нам третий способ перемещения данных из одного места в другое:

```
std::vector<std::string> input = {"hello", "world"};
std::vector<std::string> output(2);

// Третий способ: использование std::transform
std::transform(
  input.begin(),
  input.end(),
  output.begin(),
  std::move<std::string&>
);
```

Но я не рекомендую пользоваться им! Самый большой и самый красный из его красных флажков – явная специализация шаблона `std::move`. Всякий раз, когда присутствует явная специализация – эти угловые скобки после имени шаблона, – это почти всегда верный признак очень чувствительного и хрупкого кода. Опытным читателям может доставить удовольствие самим понять, как компилятор определяет типы двух `std::move` в моем примере; напомню, что один из них определен в заголовке `<utility>`, а другой – в `<algorithm>`.

Диапазонные алгоритмы только для записи

Эту главу мы начали с исследования таких алгоритмов, как `std::find`, которые выполняют обход диапазона, читая его элементы, не изменяя их. Возможно, вы удивитесь, узнав, что обратная операция также имеет смысл: существует целое семейство стандартных алгоритмов, которые выполняют обход диапазона и *изменяют* элементы, не читая их!

`std::fill(a,b,v)` делает то, о чем говорит его имя: заполняет (fill) каждый элемент в заданном диапазоне [a,b) копией значения `v`.

`std::iota(a,b,v)` – более интересный алгоритм: он заполняет элементы в заданном диапазоне копиями `++v`. То есть вызов `std::iota(a,b,42)` запишет в `a[0]` значение 42, в `a[1]` – значение 43, в `a[2]` – значение 44 и так далее, вплоть до `b`. Такое забавное название алгоритма уходит корнями в язык программирования APL, где имелась функция с именем *ι* (то есть буква «йота» греческого алфавита), выполняющая эту операцию. Другой интересный факт об этом алгоритме: по какой-то непонятной причине его определение попало в стандартный заголовок `<numeric>` вместо `<algorithm>`. Это довольно странно.

`std::generate(a,b,g)` – еще более интересный алгоритм: он заполняет элементы в заданном диапазоне результатами последовательных вызовов `g()`, то есть:

```
template<class FwdIt, class T>
void fill(FwdIt first, FwdIt last, T value) {
  while (first != last) {
    *first = value;
    ++first;
  }
}

template<class FwdIt, class T>
void iota(FwdIt first, FwdIt last, T value) {
  while (first != last) {
    *first = value;
    ++value;
    ++first;
  }
}

template<class FwdIt, class G>
```

```
void generate(FwdIt first, FwdIt last, G generator) {
  while (first != last) {
    *first = generator();
    ++first;
  }
}
```

Вот пример использования каждого из этих стандартных алгоритмов для заполнения строками с разным содержимым. Проверьте себя: понимаете ли вы, почему каждый вызов производит именно такой вывод? Пример, который я выбрал для `std::iota`, особенно интересен (но вряд ли будет полезен в практическом коде):

```
std::vector<std::string> v(4);

std::fill(v.begin(), v.end(), "hello");
assert(v[0] == "hello");
assert(v[1] == "hello");
assert(v[2] == "hello");
assert(v[3] == "hello");

std::iota(v.begin(), v.end(), "hello");
assert(v[0] == "hello");
assert(v[1] == "ello");
assert(v[2] == "llo");
assert(v[3] == "lo");

std::generate(v.begin(), v.end(), [i=0]() mutable {
  return ++i % 2 ? "hello" : "world";
});
assert(v[0] == "hello");
assert(v[1] == "world");
assert(v[2] == "hello");
assert(v[3] == "world");
```

Алгоритмы, влияющие на жизненный цикл объектов

Заголовок `<memory>` определяет семейство алгоритмов с малопонятными именами, такими как `std::uninitialized_copy`, `std::uninitialized_default_construct` и `std::destroy` (полный список можно найти в онлайн-справочнике, таком как cppreference.com). Рассмотрим следующий алгоритм, который использует неявный вызов деструктора для уничтожения элементов в диапазоне:

```
template<class T>
void destroy_at(T *p)
{
```

```
  p->~T();
}

template<class FwdIt>
void destroy(FwdIt first, FwdIt last)
{
  for ( ; first != last; ++first) {
    std::destroy_at(std::addressof(*first));
  }
}
```

Обратите внимание на небольшую, удобную вспомогательную функцию `std::addressof(x)`, которая возвращает адрес своего параметра; она действует в точности так же, как `&x`, кроме редких случаев, когда `x` имеет тип класса, который по-садистски перегружает `operator&`.

А теперь взглянем на следующий алгоритм, который использует явный синтаксис размещающей формы `new` (placement-new) для «конструирования копированием в» элементы диапазона (обратите внимание, как он аккуратно прибирает за собой, если в ходе копирования возникло исключение). Очевидно, что этот алгоритм не должен использоваться с диапазонами, в которых уже имеются существующие элементы; поэтому следующий пример выглядит надуманным:

```
template<class It, class FwdIt>
FwdIt uninitialized_copy(It first, It last, FwdIt out)
{
  using T = typename std::iterator_traits<FwdIt>::value_type;
  FwdIt old_out = out;
  try {
    while (first != last) {
      ::new (static_cast<void*>(std::addressof(*out))) T(*first);
      ++first;
      ++out;
    }
    return out;
  } catch (...) {
    std::destroy(old_out, out);
    throw;
  }
}

void test()
{
  alignas(std::string) char b[5 * sizeof (std::string)];
  std::string *sb = reinterpret_cast<std::string *>(b);

  std::vector<const char *> vec = {"quick", "brown", "fox"};

  // Сконструировать три std::string
```

```
    auto end = std::uninitialized_copy(vec.begin(), vec.end(), sb);

    assert(end == sb + 3);

    // Уничтожить три std::string.
    std::destroy(sb, end);
}
```

Больше о практическом назначении этих алгоритмов вы узнаете в главе 4 «Зоопарк контейнеров», когда пойдет разговор о `std::vector`.

Наш первый перестановочный алгоритм: std::sort

Все алгоритмы, рассматривавшиеся до сих пор, просто выполняют обход заданного диапазона по порядку, линейно, от одного элемента к следующему. Наше следующее семейство алгоритмов проявляет иное поведение. Вместо этого они извлекают значения из элементов в заданном диапазоне и перетасовывают их так, что те же значения сохраняются в диапазоне, но следуют уже в другом порядке. В математике такая операция называется *перестановкой* (permutation).

Проще всего, пожалуй, описать перестановочный алгоритм `std::sort(a,b)`. Как следует из его имени, он сортирует данные в указанном диапазоне так, что наименьшие элементы оказываются в начале, а наибольшие – в конце. Для определения «наименьшего» элемента `std::sort(a,b)` использует `operator<`.

Если нужен какой-то другой порядок, можно попробовать перегрузить `operator<` так, чтобы он возвращал `true` при иных условиях, но в такой ситуации, вероятно, удобнее использовать трехаргументную версию алгоритма, `std::sort(a,b,cmp)`. Третий аргумент должен быть *компаратором*; то есть функцией, функтором или лямбда-выражением, возвращающей(им) `true`, когда первый аргумент «меньше» второго. Например:

```
std::vector<int> v = {3, 1, 4, 1, 5, 9};
std::sort(v.begin(), v.end(), [](auto&& a, auto&& b) {
    return a % 7 < b % 7;
});
assert((v == std::vector{1, 1, 9, 3, 4, 5}));
```

Обратите внимание, что я тщательно подобрал лямбда-выражение в этом примере, чтобы сортировка массива выполнялась детерминированным способом. Если бы я выбрал функцию `(a % 6 < b % 6)`, тогда были бы возможны два варианта результата: `{1, 1, 3, 9, 4, 5}` или `{1, 1, 9, 3, 4, 5}`. Стандартный алгоритм `sort` не дает никаких гарантий относительно взаиморасположения элементов, *равных* с точки зрения функции сравнения!

Исправить эту проблему (если считать ее проблемой) можно, воспользовавшись алгоритмом `with  std::stable_sort` вместо `std::sort`. Он может

оказаться немного медленнее, но в случае равенства элементов гарантирует сохранность исходного порядка их расположения, то есть в примере с лямбда-выражением (`a % 6 < b % 6`) мы получим результат {`1, 1, 3, 9, 4, 5`}, потому что в оригинальном (несортированном) векторе элемент 3 предшествует элементу 9.

Но это еще не самая худшая ситуация с `sort` и `stable_sort` – представьте, что я выбрал функцию сравнения (`a % 6 < b`). Тогда некоторые пары элементов `x`, `y` могли бы одновременно соответствовать двум условиям: `x < y` и `y < x`! (Одна из таких пар в оригинальном векторе: 5 и 9.) В данном случае ничто не может спасти нас; мы использовали «функцию сравнения», которая просто *не* является функцией сравнения! Это является нарушением предварительных условий `std::sort`, просто как если бы мы передали пустой указатель. Сортируя массивы, убедитесь, что сортируете с применением функции сравнения, имеющей смысл!

Обмен местами, обратное упорядочение и разделение

Библиотека STL содержит удивительно большое количество перестановочных алгоритмов, кроме `std::sort`. Многие из них можно считать «строительными блоками», которые реализуют малую часть возможностей универсального алгоритма сортировки.

`std::swap(a,b)` – самый простой строительный блок; он просто принимает два аргумента и «меняет» их местами, точнее, меняет местами их значения. Он реализован в терминах конструктора перемещения для данного типа и оператора перемещения. В действительности `swap` стоит немного особняком в ряду стандартных алгоритмов, потому что реализует очень простую операцию и потому что почти всегда существует более быстрый способ поменять местами два произвольных объекта вместо эквивалентной последовательности `temp = a; a = b; b = temp;`. Обычно типы в стандартной библиотеке (такие как `std::vector`) реализуют свою функцию-член `swap` (например, `a.swap(b)`) и добавляют перегруженную версию `swap` в свое пространство имен – то есть, реализуя свой тип `my::obj`, мы можем добавить перегруженную версию в пространство имен `my` – так, что `swap(a,b)` для данного типа будет вызывать `a.swap(b)` вместо трех операций перемещения. Например:

```
namespace my {
  class obj {
    int v;
  public:
    obj(int value) : v(value) {}

    void swap(obj& other) {
      using std::swap;
      swap(this->v, other.v);
```

```
    }
  };

  void swap(obj& a, obj& b) {
    a.swap(b);
  }
} // namespace my

void test()
{
  int i1 = 1, i2 = 2;
  std::vector<int> v1 = {1}, v2 = {2};
  my::obj m1 = 1, m2 = 2;
  using std::swap;
  swap(i1, i2); // вызовет std::swap<int>(int&, int&)
  swap(v1, v2); // вызовет std::swap(vector&, vector&)
  swap(m1, m2); // вызовет my::swap(obj&, obj&)
}
```

Теперь, после знакомства с алгоритмом swap и двунаправленными итераторами, мы можем сконструировать `std::reverse(a,b)`, перестановочный алгоритм, который просто меняет порядок следования элементов в диапазоне на обратный, меняя местами первый элемент с последним, второй с предпоследним и т. д. Одним из типичных применений `std::reverse` является обращение порядка следования больших фрагментов строк – например, для обращения порядка следования слов в предложении:

```
void reverse_words_in_place(std::string& s)
{
  // Сначала обратить порядок следования символов во всей строке.
  std::reverse(s.begin(), s.end());

  // Затем вернуть обратно порядок символов в отдельных словах.
  for (auto it = s.begin(); true; ++it) {
    auto next = std::find(it, s.end(), ' ');
    // Обратить порядок следования символов в этом слове.
    std::reverse(it, next);
    if (next == s.end()) {
      break;
    }
    it = next;
  }
}

void test()
{
  std::string s = "the quick brown fox jumps over the lazy dog";
  reverse_words_in_place(s);
  assert(s == "dog lazy the over jumps fox brown quick the");
}
```

Небольшая корректировка реализации `std::reverse` дает нам еще один строительный блок, а именно `std::partition`. В то время как `std::reverse` выполняет обход диапазона сразу с двух концов и безусловно меняет местами каждую пару элементов, алгоритм `std::partition` меняет элементы местами, только если они стоят «не по порядку», согласно заданной функции-предикату. Следующий пример переносит все четные элементы в начало диапазона, а все нечетные – в конец. Если бы мы использовали `std::partition` для реализации процедуры быстрой сортировки, мы могли бы перенести элементы *меньше опорного значения* в начало диапазона, а элементы *больше опорного значения* – в его конец:


```
template<class BidirIt>
void reverse(BidirIt first, BidirIt last)
{
  while (first != last) {
    --last;
    if (first == last) break;
    using std::swap;

    swap(*first, *last);
    ++first;
  }
}

template<class BidirIt, class Unary>
auto partition(BidirIt first, BidirIt last, Unary p)
{
  while (first != last && p(*first)) {
    ++first;
  }
  while (first != last) {
    do {
      --last;
    } while (last != first && !p(*last));
    if (first == last) break;
    using std::swap;

    swap(*first, *last);
    do {
      ++first;
    } while (first != last && p(*first));
  }
  return first;
}

void test()
{
  std::vector<int> v = {3, 1, 4, 1, 5, 9, 2, 6, 5};
  auto it = std::partition(v.begin(), v.end(), [](int x) {
    return x % 2 == 0;
```

```
  });
assert(it == v.begin() + 3);
assert((v == std::vector{6, 2, 4, 1, 5, 9, 1, 3, 5}));
}
```

Обратите внимание на один интересный аспект в предыдущем коде: реализации `reverse` и `partition` практически идентичны! Единственное отличие – `partition` содержит неуклюжий цикл `do-while`, вместо которого в `reverse` используются простые операции инкремента и декремента.

Отметьте также, что первый цикл `do-while` в `partition` эквивалентен стандартному алгоритму, который мы уже видели; а именно `std::find_if_not`. И второй цикл `do-while` можно считать своеобразным эквивалентом `std::find_if`..., только он выполняется в обратном направлении, а не в прямом! К сожалению для нас, нет такого алгоритма, как `std::rfind_if`. Но – как вы уже понимаете – стандартная библиотека не собирается оставлять нас в беде.

Нам нужно нечто, обладающее поведением итератора с точки зрения `std::find_if`, но выполняющее итерации «в обратном направлении». Стандартная библиотека реализует именно такое нечто в форме адаптера `std::reverse_iterator`. Я не буду показывать его реализацию, а если вам нужно вспомнить, как такой адаптер можно было бы реализовать, вернитесь к главе 2 «Итераторы и диапазоны». Достаточно будет сказать, что объект `std::reverse_iterator<FwdIt>` обертывает объект `FwdIt` и действует подобно ему, кроме того что когда к обертке применяется операция инкремента, она выполняет декремент обернутого объекта, и наоборот. Соответственно, мы можем реализовать `partition` в терминах `reverse_iterator`, как показано ниже:

```
// Сокращенно от "reversing" и "unreversing".
template<class It>
auto rev(It it) {
  return std::reverse_iterator(it);
};

template<class InnerIt>
auto unrev(std::reverse_iterator<InnerIt> it) {
  return it.base();
}

template<class BidirIt, class Unary>
auto partition(BidirIt first, BidirIt last, Unary p)
{
  first = std::find_if_not(first, last, p);

  while (first != last) {
    last = unrev(std::find_if(rev(last), rev(first), p));
    if (first == last) break;
    using std::swap;

    swap(*first++, *--last);
```

```
    first = std::find_if_not(first, last, p);
  }
  return first;
}
```

Иногда полезно также иметь возможность разбить диапазон, не меняя относительного порядка следования элементов в разделах. Для таких случаев можно использовать `std::stable_partition(a,b,p)` (но в `stable_partition` есть одна загвоздка, как рассказывается в разделе «Слияние и сортировка слиянием»: он может выделять память с помощью оператора `new`).

Существует несколько неперестановочных алгоритмов, которые тоже связаны с разделами: `std::is_partitioned(a,b,p)` возвращает `true`, если заданный диапазон уже поделен на разделы в соответствии с предикатом `p` (то есть все элементы, удовлетворяющие предикату `p`, находятся в начале, а все остальные – в конце).

`std::partition_point(a,b,p)` использует поиск методом дихотомии, чтобы найти первый элемент в диапазоне, уже разбитом на разделы, который не удовлетворяет `p`.

`std::partition_copy(a,b,ot,of,p)` копирует каждый элемент из диапазона `[a,b)` в тот или другой итератор вывода: `*ot++ = e` – для элементов с `p(e) == true` и `*of++ = e` – для элементов с `p(e) == false`.

Кстати, если вывод должен осуществляться только в одну из двух последовательностей, можно использовать `std::copy_if(a,b,ot,p)` или `std::remove_copy_if(a,b,of,p)` соответственно.

Ротация и перестановка

Вспомните наш код в разделе «Обмен местами, обратное упорядочение и разделение», меняющий порядок следования слов в предложении на обратный. В случае с «предложением», содержащим только два слова, на алгоритм обращения можно посмотреть иначе: как на алгоритм *циклической ротации* элементов в заданном диапазоне. `std::rotate(a,mid,b)` выполняет ротацию элементов в диапазоне `[a,b)` так, что элемент в позиции `mid` оказывается в позиции `a` (и возвращает итератор, указывающий на элемент, значение которого прежде находилось в позиции `a`):

```
template<class FwdIt>
FwdIt rotate(FwdIt a, FwdIt mid, FwdIt b)
{
  auto result = a + (b - mid);

  // Сначала обратим весь диапазон.
  std::reverse(a, b);

  // Затем обратим каждый из сегментов.
  std::reverse(a, result);
```

```
  std::reverse(result, b);

  return result;
}

void test()
{
  std::vector<int> v = {1, 2, 3, 4, 5, 6};
  auto five = std::find(v.begin(), v.end(), 5);
  auto one = std::rotate(v.begin(), five, v.end());
  assert((v == std::vector{5, 6, 1, 2, 3, 4}));
  assert(*one == 1);
}
```

Еще один перестановочный алгоритм, который иногда может оказаться полезным: `std::next_permutation(a,b)`. Вызов этой функции в цикле позволяет выполнить обход всех возможных перестановок из *n* элементов, что может пригодиться для реализации решения «в лоб» «Задачи коммивояжера»[2] (с небольшим количеством узлов):

```
std::vector<int> p = {10, 20, 30};
std::vector<std::vector<int>> results;

// Перебрать перестановки из этих трех элементов.
for (int i=0; i < 6; ++i) {
  results.push_back(p);
  std::next_permutation(p.begin(), p.end());
}

assert((results == std::vector<std::vector<int>>{
  {10, 20, 30},
  {10, 30, 20},
  {20, 10, 30},
  {20, 30, 10},
  {30, 10, 20},
  {30, 20, 10},
}));
```

Обратите внимание, что `next_permutation` использует идею отношения «меньше, чем» для определения, является одна перестановка лексикографически «меньше» другой; например, `{20, 10, 30}` «меньше», чем `{20, 30, 10}`, потому что `10` меньше `30`. Есть также версия `next_permutation` с компаратором: `std::next_permutation(a,b,cmp)`, а кроме того, имеются алгоритмы `std::prev_permutation(a,b)` и `std::prev_permutation(a,b,cmp)`, которые считаются лексикографически «нисходящими», а не «восходящими».

Кстати, для подобного лексикографического сравнения двух последовательностей можно применить алгоритм `std::mismatch`, описанный в разделе

[2] https://ru.wikipedia.org/wiki/Задача_коммивояжёра – *Прим. перев.*

«Диапазонные алгоритмы только для чтения», или просто использовать стандартный алгоритм `std::lexicographical_compare(a,b,c,d)`.

Кучи и пирамидальная сортировка

`std::make_heap(a,b)` (или его версия с компаратором `std::make_heap(a,b,cmp)`) принимает диапазон несортированных элементов и размещает их в порядке, удовлетворяющем *свойству невозрастания пирамиды* (max-heap property): в массиве со свойством невозрастания пирамиды каждый элемент с индексом i из диапазона будет, по крайней мере, столь же большим, как любой из элементов с индексами $2i + 1$ и $2i + 2$. Это означает, что элемент с самым большим значением будет иметь индекс 0.

`std::push_heap(a,b)` (или его версия с компаратором) предполагает, что диапазон `[a,b-1)` уже обладает свойством невозрастания пирамиды. Он берет элемент в позиции `b[-1]` и начинает передвигать его в начало диапазона, пока не восстановится свойство невозрастания пирамиды для всего диапазона `[a,b)`. Обратите внимание, что `make_heap` можно реализовать как простой цикл, многократно вызывающий `std::push_heap(a,++b)`.

`std::pop_heap(a,b)` (или его версия с компаратором) предполагает, что диапазон `[a,b)` уже обладает свойством невозрастания пирамиды. Он меняет местами `a[0]` и `b[-1]`, в результате чего наибольший элемент теперь оказывается в *конце* диапазона, а не в начале; и затем начинает переставлять местами `a[0]` с каждым следующим элементом, перемещая его в конец, пока не восстановится свойство невозрастания пирамиды. После вызова `pop_heap(a,b)` наибольший элемент будет находиться в `b[-1]`, а диапазон `[a, b-1)` – соответствовать свойству невозрастания пирамиды.

`std::sort_heap(a,b)` (или его версия с компаратором) принимает диапазон, обладающий свойством невозрастания пирамиды, и переставляет элементы в порядке сортировки, в цикле вызывая `std::pop_heap(a, b--)`.

Используя эти строительные блоки, мы можем реализовать классический алгоритм «пирамидальной» сортировки. Функцию `std::sort` из стандартной библиотеки вполне можно было бы реализовать так (но на практике она обычно реализует какой-нибудь гибридный алгоритм, такой как «интроспективная сортировка»):

```
template<class RandomIt>
void push_heap(RandomIt a, RandomIt b)
{
  auto child = ((b-1) - a);
  while (child != 0) {
    auto parent = (child - 1) / 2;
    if (a[child] < a[parent]) {
      return; // свойство невозрастания пирамиды восстановлено
    }
    std::iter_swap(a+child, a+parent);
    child = parent;
```

```
  }
}

template<class RandomIt>
void pop_heap(RandomIt a, RandomIt b)
{
  using DistanceT = decltype(b - a);

  std::iter_swap(a, b-1);

  DistanceT parent = 0;
  DistanceT new_heap_size = ((b-1) - a);

  while (true) {
    auto leftchild = 2 * parent + 1;
    auto rightchild = 2 * parent + 2;
    if (leftchild >= new_heap_size) {
      return;
    }
    auto biggerchild = leftchild;
    if (rightchild < new_heap_size && a[leftchild] < a[rightchild]) {
      biggerchild = rightchild;
    }
    if (a[biggerchild] < a[parent]) {
      return; // свойство невозрастания пирамиды восстановлено
    }
    std::iter_swap(a+parent, a+biggerchild);
    parent = biggerchild;
  }
}

template<class RandomIt>
void make_heap(RandomIt a, RandomIt b)
{
  for (auto it = a; it != b; ) {
    push_heap(a, ++it);
  }
}

template<class RandomIt>
void sort_heap(RandomIt a, RandomIt b)
{
  for (auto it = b; it != a; --it) {
    pop_heap(a, it);
  }
}

template<class RandomIt>
void sort(RandomIt a, RandomIt b)
{
  make_heap(a, b);
```

```
  sort_heap(a, b);
}
```

Другие применения `push_heap` и `pop_heap` мы увидим в главе 4 «Зоопарк контейнеров», когда будем говорить о `std::priority_queue`.

Слияние и сортировка слиянием

Коль скоро мы занялись обсуждением алгоритмов сортировки, реализуем сортировку другим способом! `std::inplace_merge(a,mid,b)` принимает единственный диапазон `[a,b)`, уже отсортированный эквивалентами `std::sort(a,mid)` и `std::sort(mid,b)`, и объединяет два поддиапазона в один отсортированный диапазон. Этот строительный блок можно использовать для реализации классического алгоритма сортировки слиянием:

```
template<class RandomIt>
void sort(RandomIt a, RandomIt b)
{
  auto n = std::distance(a, b);
  if (n >= 2) {
    auto mid = a + n/2;
    std::sort(a, mid);
    std::sort(mid, b);
    std::inplace_merge(a, mid, b);
  }
}
```

Но будьте внимательны! Имя `inplace_merge`, как может показаться, подразумевает, что слияние (или объединение) происходит «на месте» (in-place) и не требуется создавать буфер с дополнительным пространством; но в действительности это не так. На самом деле функция `inplace_merge` сама выделяет память для буфера с помощью оператора `new`. Если вы пишете программу для работы в окружении, где выделение динамической памяти является большой проблемой, избегайте использования `inplace_merge` как чумы.

К другим стандартным алгоритмам, которые могут выделять временные буферы в динамической памяти, относятся `std::stable_sort` и `std::stable_partition`.

Алгоритм слияния `std::merge(a,b,c,d,o)` не выделяет дополнительной памяти; он принимает две пары итераторов, представляющих диапазоны `[a,b)` и `[c,d)`, и объединяет их в один выходной диапазон, определяемый итератором `o`.

Поиск и вставка в сортированный массив с std::lower_bound

После сортировки диапазона данных появляется возможность выполнять поиск с использованием процедуры поиска методом дихотомии, который действует существенно быстрее метода линейного поиска. Стандартный ал-

горитм, реализующий поиск методом дихотоимии, называется `std::lower_bound(a,b,v)`:

```
template<class FwdIt, class T, class C>
FwdIt lower_bound(FwdIt first, FwdIt last, const T& value, C lessthan)
{
  using DiffT = typename std::iterator_traits<FwdIt>::difference_type;
  FwdIt it;
  DiffT count = std::distance(first, last);

  while (count > 0) {
    DiffT step = count / 2;
    it = first;
    std::advance(it, step);
    if (lessthan(*it, value)) {
      ++it;
      first = it;
      count -= step + 1;
    } else {
      count = step;
    }
  }
  return first;
}

template<class FwdIt, class T>
FwdIt lower_bound(FwdIt first, FwdIt last, const T& value)
{
  return std::lower_bound(first, last, value, std::less<>{});
}
```

Эта функция возвращает итератор, указывающий на первый элемент в диапазоне, который *не меньше* заданного значения `v`. Если в диапазоне имеется экземпляр значения `v`, она вернет итератор, указывающий на него (на самом деле она вернет итератор, указывающий на *первый* такой экземпляр в диапазоне). Если в диапазоне нет такого экземпляра, функция вернет итератор на место, где такой экземпляр `v` должен был бы находиться.

Значение, возвращаемое функцией `lower_bound`, можно использовать как аргумент для функции `vector::insert`, чтобы вставить `v` в сортированный вектор в соответствующее место, не нарушив порядка сортировки:

```
std::vector<int> vec = {3, 7};
for (int value : {1, 5, 9}) {
  // Найти место для вставки...
  auto it = std::lower_bound(vec.begin(), vec.end(), value);
  // ...и вставить значение value.
  vec.insert(it, value);
}
// Вектор остается отсортированным.
assert((vec == std::vector{1, 3, 5, 7, 9}));
```

Похожая функция `std::upper_bound(a,b,v)` возвращает итератор, указывающий на первый элемент в диапазоне, который *больше* заданного значения `v`. Если `v` отсутствует в указанном диапазоне, `std::upper_bound` вернет то же значение, что и функция `std::lower_bound`. Но если `v` присутствует в диапазоне, тогда `lower_bound` вернет итератор, указывающий на первый экземпляр `v` в диапазоне, а `upper_bound` – на элемент, следующий за *последним* экземпляром. Иными словами, объединив эти две функции, можно получить полуоткрытый диапазон `[lower, upper)`, содержащий только значения `v`:

```
std::vector<int> vec = {2, 3, 3, 3, 4};
auto lower = std::lower_bound(vec.begin(), vec.end(), 3);

// Первый подход:
// интерфейс upper_bound идентичен интерфейсу lower_bound.
auto upper = std::upper_bound(vec.begin(), vec.end(), 3);

// Второй подход:
// Во второй раз необязательно выполнять поиск по всему массиву.
auto upper2 = std::upper_bound(lower, vec.end(), 3);
assert(upper2 == upper);

// Третий подход:
// Линейное сканирование от нижней границы lower может оказаться быстрее,
// чем поиск методом дихотомии, если диапазон имеет очень большой размер.
auto upper3 = std::find_if(lower, vec.end(), [](int v) {
  return v != 3;
});
assert(upper3 == upper);

// При любом подходе получаются одни и те же результаты.
assert(*lower >= 3);
assert(*upper > 3);
assert(std::all_of(lower, upper, [](int v) { return v == 3; }));
```

Мы рассмотрели поиск и вставку значений в сортированный массив. А как выполнить удаление?

Удаление из сортированного массива с std::remove_if

До сих пор, обсуждая стандартные обобщенные алгоритмы, мы не касались вопроса удаления элементов из диапазона. Это объясняется тем, что фундаментально «диапазон» доступен только для чтения: мы можем изменять значения элементов в диапазоне, но с помощью стандартных алгоритмов нельзя укоротить или удлинить сам диапазон. В разделе «Манипулирование данными с std::copy» мы использовали `std::copy` для «вставки в» вектор с именем `dest`, но это не был алгоритм `std::copy`, выполняющий вставку; это был объект

`std::back_insert_iterator`, хранящий ссылку на базовый контейнер и способный выполнить вставку в контейнер. Алгоритм `std::copy` не принимает параметры `dest.begin()` и `dest.end()`; вместо этого он принимает особый объект `std::back_inserter(dest)`.

Итак, как же нам удалить элемент из диапазона? А никак – это невозможно. Все, что мы можем, – это удалить элемент из контейнера средствами самого контейнера, а алгоритмы из STL не работают с контейнерами. Поэтому нам нужно найти способ переупорядочить значения в диапазоне так, чтобы «удаленные» элементы оказались где-то в предопределенном месте внутри контейнера, чтобы потом их можно было быстро удалить (используя другие средства, отличные от алгоритмов STL).

Мы уже видели одно из возможных решений:

```
std::vector<int> vec = {1, 3, 3, 4, 6, 8};

// Разбить вектор так, чтобы все элементы, не равные 3, оказались в начале
// а все элементы, равные 3, оказались в конце.
auto first_3 = std::stable_partition(
  vec.begin(), vec.end(), [](int v){ return v != 3; }
);

assert((vec == std::vector{1, 4, 6, 8, 3, 3}));

// Теперь можно удалить "хвост" вектора.
vec.erase(first_3, vec.end());

assert((vec == std::vector{1, 4, 6, 8}));
```

Но здесь делается много ненужной работы (не забывайте, что `stable_partition` – один из алгоритмов STL, которые выделяют буферы в динамической памяти!). На самом деле алгоритм, нужный нам, выглядит намного проще:

```
template<class FwdIt, class T>
FwdIt remove(FwdIt first, FwdIt last, const T& value)
{
  auto out = std::find(first, last, value);
  if (out != last) {
    auto in = out;
    while (++in != last) {
      if (*in == value) {
        // этот элемент нас не интересует
      } else {
        *out++ = std::move(*in);
      }
    }
  }
  return out;
}

void test()
```

```
{
  std::vector<int> vec = {1, 3, 3, 4, 6, 8};

  // Разбить вектор так, чтобы все элементы, не равные 3, оказались в начале
  auto new_end = std::remove(
    vec.begin(), vec.end(), 3
  );

  // std::remove_if не сохраняет "удаленные" элементы.
  assert((vec == std::vector{1, 4, 6, 8, 6, 8}));

  // Теперь можно удалить "хвост" вектора.
  vec.erase(new_end, vec.end());

  assert((vec == std::vector{1, 4, 6, 8}));

  // Или выполнить оба шага в одной строке.
  // Это идиома "стереть-удалить":
  vec.erase(
    std::remove(vec.begin(), vec.end(), 3),
    vec.end()
  );

  // Но если массив очень большой и известно, что он отсортирован,
  // тогда, возможно, поиск элементов для удаления лучше производить
  // методом дихотомии
  // Здесь также происходит "сдвиг вниз", но уже
  // внутри vector::erase, а не std::remove.
  auto first = std::lower_bound(vec.begin(), vec.end(), 3);
  auto last = std::upper_bound(first, vec.end(), 3);
  vec.erase(first, last);
}
```

`std::remove(a,b,v)` удаляет из диапазона `[a,b)` все значения, равные `v`. При этом не требуется, чтобы данные в диапазоне были отсортированы, – но `remove` сохраняет исходный порядок, «сдвигая вниз» неудаленные элементы для заполнения образовавшихся «пустот» в диапазоне. Если `remove` удалит *k* элементов из диапазона, тогда по возвращении из нее эти *k* перемещенных элементов окажутся в конце диапазона, а возвращаемое значение `remove` будет содержать итератор, указывающий на первый такой перемещенный элемент.

`std::remove_if(a,b,p)` удаляет все элементы, удовлетворяющие заданному предикату `p`; то есть она удалит все элементы `e`, для которых выполняется условие `p(e) == true`. Так же как `remove`, алгоритм `remove_if` сдвигает элементы вниз, чтобы заполнить образующиеся пробелы, и возвращает итератор, указывающий на первый перемещенный элемент.

Типичная идиома удаления элементов из последовательного контейнера известна как *идиома стирания-удаления*, потому что предполагает возможность передачи возвращаемого значения непосредственно в функцию-член `.erase()` контейнера.

Другой алгоритм из стандартной библиотеки, который использует идиому стирания-удаления, – функция `std::unique(a,b)`. Она принимает диапазон и из каждого набора одинаковых элементов, следующих друг за другом, удаляет все, кроме первого. По аналогии с `std::remove`, входной диапазон необязательно должен быть отсортирован; алгоритм сохраняет начальный порядок элементов:

```
std::vector<int> vec = {1, 2, 2, 3, 3, 3, 1, 3, 3};

vec.erase(
  std::unique(vec.begin(), vec.end()),
  vec.end()
);

assert((vec == std::vector{1, 2, 3, 1, 3}));
```

Наконец, обратите внимание, что часто есть средство лучше, чем `std::remove`, обычно в виде функции-члена `erase` контейнера (например, в следующей главе мы увидим, что `std::list::erase` может действовать намного быстрее, чем идиома стирания-удаления с `std::list`). И даже если удаление происходит из вектора, в котором порядок следования элементов не имеет значения, все равно обычно есть лучшее решение, например напоминающее следующий алгоритм `unstable_remove`, предложенный для стандартизации в будущем, но (на момент написания этих строк) еще не включенный в:

```
namespace my {
  template<class BidirIt, class T>
  BidirIt unstable_remove(BidirIt first, BidirIt last, const T& value)
  {
    while (true) {
      // Найти первый экземпляр значения value...
      first = std::find(first, last, value);
      // ...и последний экземпляр значения, не равного value...
      do {
        if (first == last) {
          return last;
        }
        --last;
      } while (*last == value);
      // ...и переместить последний на место первого.
      *first = std::move(*last);
      // Смыть и повторить.
      ++first;
    }
  }
} // namespace my

void test()
{
```

```
  std::vector<int> vec = {4, 1, 3, 6, 3, 8};

  vec.erase(
    my::unstable_remove(vec.begin(), vec.end(), 3),
    vec.end()
  );

  assert((vec == std::vector{4, 1, 8, 6}));
}
```

В следующей главе мы рассмотрим контейнеры – ответ STL на вопрос: «Где хранятся все эти элементы?».

Итоги

Стандартная библиотека шаблонов (Standard Template Library) включает все (почти) необходимые алгоритмы. Если вы решаете какую-то алгоритмическую задачу, загляните сначала в STL!

Алгоритмы в STL работают с полуоткрытыми диапазонами, определяемыми парами итераторов. Будьте осторожны при работе с полуторадиапазонными алгоритмами.

Сравнение и сортировка алгоритмами STL по умолчанию выполняются с использованием `operator<`, но при этом есть возможность явно передать двухаргументный «компаратор». Если потребуется выполнить нетривиальную операцию с целым диапазоном данных, не забывайте, что в STL есть непосредственная (`std::move`, `std::transform`) и косвенная поддержка этого, через специальный тип итератора (`std::back_inserter`, `std::istream_iterator`).

Вы должны знать, что такое «перестановка» и как реализованы стандартные перестановочные алгоритмы (`swap`, `reverse`, `rotate`, `partition`, `sort`) в терминах друг друга. Только три алгоритма в STL (`stable_sort`, `stable_partition`, `inplace_merge`) могут за кулисами выделять динамическую память для буфера; если использование динамической памяти нежелательно, избегайте этих алгоритмов.

Используйте идиому стирания-удаления для поддержания порядка сортировки элементов в контейнере, даже при удалении элементов из него. Используйте что-нибудь вроде `my::unstable_remove`, если сохранность порядка сортировки не имеет значения. Используйте функцию-член `.erase()` при работе с контейнерами, поддерживающими ее.

Глава 4

Зоопарк контейнеров

В предыдущих главах мы познакомились с итераторами и диапазонами (глава 2 «Итераторы и диапазоны») и разнообразными обобщенными алгоритмами в стандартной библиотеке, которые оперируют диапазонами данных, представленными парами итераторов (глава 3 «Алгоритмы с парами итераторов»). В этой главе мы посмотрим, где фактически хранятся эти данные. То есть, узнав все о выполнении итераций, мы оказываемся перед вопросом: как выглядит нечто, над чем выполняются итерации?

Библиотека STL дает на этот вопрос обобщенный ответ: итерации выполняются по некоторому поддиапазону элементов, находящихся в *контейнере*. Контейнер – это всего лишь класс C++ (или шаблонный класс), который по своей природе может *хранить* (или *владеть*) гомогенный диапазон элементов данных и экспортирует этот диапазон для итераций с применением обобщенных алгоритмов.

В этой главе рассматриваются следующие темы:

- понятие *владения* одного объекта другим (это ключевая разница между *контейнером* и *диапазоном*);
- контейнеры последовательностей (`array`, `vector`, `list` и `forward_list`);
- ловушки недействительных итераторов и недействительных ссылок;
- адаптеры контейнеров (`stack`, `queue` и `priority_queue`);
- ассоциативные контейнеры (`set`, `map` и им подобные);
- когда это необходимо, дается возможность передать шаблон типа *компаратора*, *хеш-функции*, *компаратора проверки равенства* или *функции распределения памяти*.

Понятие владения

Когда мы говорим, что объект `A` *владеет* объектом `B`, мы имеем в виду, что объект `A` управляет жизненным циклом объекта `B` – то есть управляет созданием, копированием, перемещением и уничтожением объекта `B`. Пользователь объекта `A` может (и должен) «забыть об» управлении объектом `B` (например, явными вызовами `delete B`, `fclose(B)` и т. д.).

Самый простой пример «владения» объекта B объектом A – когда B является переменной-членом в объекте A. Например:

```
struct owning_A {
  B b_;
};

struct non_owning_A {
  B& b_;
};

void test()
{
  B b;

  // a1 владеет [копией] b.
  owning_A a1 { b };

  // a2 хранит лишь ссылку на b;
  // a2 не владеет объектом b.
  non_owning_A a2 { b };
}
```

Также A может хранить указатель на B с соответствующим кодом в ~A() (и, если необходимо, в операциях копирования и перемещения A) для освобождения ресурсов, связанных с этим указателем:

```
struct owning_A {
  B *b_;

  explicit owning_A(B *b) : b_(b) {}

  owning_A(owning_A&& other) : b_(other.b_) {
    other.b_ = nullptr;
  }

  owning_A& operator= (owning_A&& other) {
    delete b_;
    b_ = other.b_;
    other.b_ = nullptr;
    return *this;
  }

  ~owning_A() {
    delete b_;
  }
};

struct non_owning_A {
  B *b_;
```

```
};

void test()
{
  B *b = new B;

  // a1 владеет *b.
  owning_A a1 { b };

  // a2 просто хранит указатель *b;
  // a2 не владеет *b.
  non_owning_A a2 { b };
}
```

Понятие *владения* неразрывно связано с известной в C++ идиомой **Resource Allocation Is Initialization** (получение ресурса есть инициализация), которое часто сокращается до **RAII**. (Этой идиоме больше подходит название «Resource Freeing Is Destruction» – освобождение ресурса есть уничтожение – но было выбрано такое название, какое выбрано.)

Цель стандартных *классов контейнеров*, дать доступ к конкретному пакету данных объекта `B`, обеспечивая при этом ясное *владение* этими данными, то есть контейнер всегда владеет своими элементами данных. (*Итераторы* или пары итераторов, напротив, определяют *диапазон*, не владеющий своими элементами данных; мы видели в главе 3 «Алгоритмы с парами итераторов», что стандартные алгоритмы, основанные на итераторах, такие как `std::remove_if`, в действительности никогда не удаляют элементов, а просто переставляют значения элементов разными способами.)

В оставшейся части главы мы исследуем разные стандартные классы контейнеров.

Простейший контейнер: std::array<T, N>

Простейший стандартный контейнер – класс `std::array<T, N>`, действующий как встроенный («C-подобный») массив. Первый шаблонный параметр в `std::array` определяет тип, а второй – количество элементов массива. Это один из немногих классов в стандартной библиотеке, где шаблонный параметр является целым числом, а не именем типа.

```
std::array<char, 3> arr {{42, 43, 44}};
```

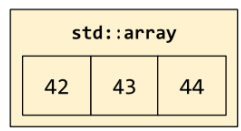


Рис. 4.1. Массив std::array

Обычные массивы в стиле C, являющиеся частью ядра языка (корнями уходящего в 1970-е!), не имеют никаких встроенных операций, требующих линейного времени на выполнение. C-массивы поддерживают индексирование с помощью `operator[]` и позволяют сравнивать их адреса, причем эти операции выполняются за постоянное время; но если вы решите присвоить все содержимое одного C-массива другому или сравнить содержимое двух массивов, то обнаружите, что это совсем не просто. Для этого вам придется использовать стандартные алгоритмы, обсуждавшиеся в главе 3 «Алгоритмы с парами итераторов», такие как `std::copy` или `std::equal` (шаблонная функция `std::swap`, будучи «алгоритмом», способна работать с C-массивами. Было бы очень странно, если бы это было не так.):


```
std::string c_style[4] = {
  "the", "quick", "brown", "fox"
};
assert(c_style[2] == "brown");
assert(std::size(c_style) == 4);
assert(std::distance(std::begin(c_style), std::end(c_style)) == 4);

// Копирование посредством operator= не поддерживается.
std::string other[4];
std::copy(std::begin(c_style), std::end(c_style), std::begin(other));

// Перестановка местами поддерживается... за линейное время, конечно.
using std::swap;
swap(c_style, other);

// Сравнение не поддерживается; необходимо использовать стандартный алгоритм.
// operator== сравнивает "неправильно" - он сравнивает адреса!
assert(c_style != other);
assert(std::equal(
  c_style, c_style + 4,
  other, other + 4
));
assert(!std::lexicographical_compare(
  c_style, c_style + 4,
  other, other + 4
));
```

`std::array` действует подобно C-массиву, но поддерживает больше синтаксического сахара. Он предлагает функции-члены `.begin()` и `.end()`; перегружает операторы `=`, `==` и `<`, чтобы они действовали более естественно. Все эти операции все еще имеют линейное время выполнения, зависящее от размера массива, потому что должны выполнять обход элементов массива, чтобы скопировать (поменять местами или сравнить) каждый элемент в отдельности.

Одна из проблем класса `std::array`, которая также встречается в некоторых других стандартных классах контейнеров, состоит в том, что при конструировании `std::array` со списком инициализаторов внутри пары фигурных ско-

бок вам придется записать две пары фигурных скобок. Первая пара определяет «внешний объект» типа `std::array<T, N>`, а вторая – «внутренний член данных» типа `T[N]`. Первое время это немного раздражает, но такой синтаксис с двойными фигурным скобками быстро входит в привычку, стоит только использовать его несколько раз:

```
std::array<std::string, 4> arr = {{
  "the", "quic", "brown", "fox"
}};
assert(arr[2] == "brown");

// Поддерживаются .begin(), .end() и .size()
assert(arr.size() == 4);
assert(std::distance(arr.begin(), arr.end()) == 4);

// Поддерживается копирование посредством operator= ...за линейное время.
std::array<std::string, 4> other;
other = arr;

// Перестановка местами также поддерживается... за линейное время.
using std::swap;
swap(arr, other);

// operator== имеет более естественное поведение: сравнивает значения!
assert(&arr != &other); // Массивы имеют разные адреса...
assert(arr == other);   // ...но лексиографически они равны.
assert(arr >= other);   // Операторы отношений тоже поддерживаются.
```

Еще одно достоинство `std::array` – эти контейнеры можно возвращать из функций, чего нельзя делать с С-массивами:

```
// Из функции нельзя вернуть простой С-массив.
// auto cross_product(const int (&a)[3], const int (&b)[3]) -> int[3];

// Но можно вернуть std::array.
auto cross_product(const std::array<int, 3>& a,
                   const std::array<int, 3>& b) -> std::array<int, 3>
{
  return {{
    a[1] * b[2] - a[2] * b[1],
    a[2] * b[0] - a[0] * b[2],
    a[0] * b[1] - a[1] * b[0],
  }};
}
```

Так как `std::array` имеет конструктор копирования и поддерживает оператор присваивания копированием, массивы этого типа можно хранить в контейнерах, например `std::vector<std::array<int, 3>>`, но с простыми массивами этот трюк `std::vector<int[3]>` не пройдет.

Однако если вы поймаете себя на том, что очень часто возвращаете массивы из функций или храните их в контейнерах, задумайтесь – действительно ли абстракция «массив» отвечает вашим потребностям. Возможно, лучше будет завернуть массив в какой-нибудь класс?

В случае с функцией `cross_product` из примера выше гораздо предпочтительнее выглядит идея заключить «массив из трех целых чисел» в класс. Это позволит не только дать элементам массива имена (`x`, `y` и `z`), но также упростить инициализацию объектов типа `Vec3` (без второй пары фигурных скобок!), и, что самое важное, пожалуй, мы можем отказаться от реализации операторов сравнения, таких как `operator<`, которые не имеют смысла в нашей математической области. Используя `std::array`, мы вынуждены учитывать тот факт, что массив `{1, 2, 3}` считается «меньше» массива `{1, 3, -9}`, но, определяя свой класс `Vec3`, можем просто забыть об `operator<` и тем самым гарантировать, что никто не сможет по ошибке использовать его в математическом контексте:


```
struct Vec3 {
  int x, y, z;
  Vec3(int x, int y, int z) : x(x), y(y), z(z) {}
};

bool operator==(const Vec3& a, const Vec3& b) {
  return std::tie(a.x, a.y, a.z) ==
    std::tie(b.x, b.y, b.z);
}

bool operator!=(const Vec3& a, const Vec3& b) {
  return !(a == b);
}

// Операторы < <= > >= не имеют смысла для

Vec3 cross_product(const Vec3& a, const Vec3& b) {
  return {
    a.y * b.z - a.z * b.y,
    a.z * b.x - a.x * b.z,
    a.x * b.y - a.y * b.x,
  };
}
```

`std::array` хранит свои элементы непосредственно в себе. Поэтому значение `sizeof (std::array<int, 100>)` равно значению `sizeof (int[100])`, которое, в свою очередь, равно `100 * sizeof (int)`. Не допускайте ошибки, пытаясь поместить гигантский массив на стек, как локальную переменную!

```
void dont_do_this()
{
  // Эта переменная займет 4 мегабайта пространства на стеке ---
  // вполне достаточно, чтобы исчерпать стек и вызвать ошибку сегментации!
  int arr[1'000'000];
```

```
}

void dont_do_this_either()
{
  // Использование std::array не решает проблемы.
  std::array<int, 1'000'000> arr;
}
```

В роли «гигантских массивов» лучше использовать следующий контейнер в нашем списке: `std::vector`.

Рабочая лошадка: std::vector<T>

`std::vector` представляет непрерывный массив элементов данных, но место для него выделяет в динамической памяти, а не на стеке. Имеет два основных преимущества перед `std::array`: во-первых, позволяет создавать по-настоящему гигантские массивы без риска переполнить стек. Во-вторых, поддерживает возможность динамического изменения размера массива – в отличие от `std::array<int, 3>`, где размер массива является неизменяемой частью типа, `std::vector<int>` не имеет ограничения на размер. Метод `.size()` вектора фактически возвращает полезную информацию о текущем состоянии вектора.

`std::vector` имеет еще один важный атрибут: *емкость*. Емкость вектора всегда больше или равна его размеру и представляет количество элементов, которое вектор мог бы хранить в данный момент, прежде чем выделить дополнительную память для базового массива:

```
std::vector<char> v {42, 43, 44};
v.reserve(8);
```

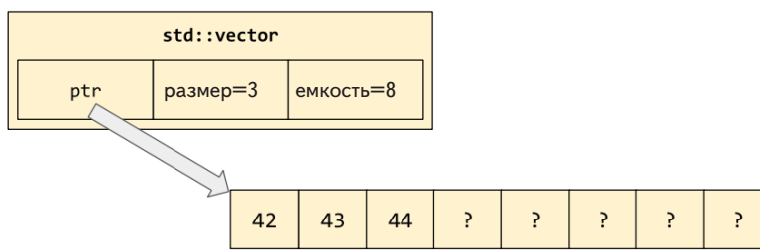


Рис. 4.2. Размер вектора и его емкость

Помимо динамического изменения размера, в остальном вектор действует подобно массиву. Так же как массивы, векторы можно копировать (за линейное время) и сравнивать (`std::vector<T>::operator<` выполняет лексикографическое сравнение операндов, используя `T::operator<`).

Вообще говоря, `std::vector` является самым часто используемым контейнером из имеющихся в стандартной библиотеке. Всякий раз, когда вам пона-

добится сохранить «много» элементов (или когда количество элементов заранее неизвестно), вашей первой мыслью должно быть использование вектора `vector`. Почему? Потому что вектор сочетает в себе преимущества контейнера динамического размера с простотой и эффективностью непрерывных массивов.

Непрерывные массивы являются наиболее эффективными структурами данных (на типичном аппаратном обеспечении), потому что обеспечивают хорошую *локальность*, также известную как *дружественность к кешу*. Перемещаясь вдоль вектора от `.begin()` до `.end()`, вы также перемещаетесь вдоль области памяти, а это значит, что процессор компьютера может с высокой точностью предсказать, какой следующий блок памяти будет просматриваться. Сравните это со связным списком, в котором обход элементов в направлении от `.begin()` к `.end()` может вовлекать разыменование указателей на блоки, расположенные в памяти не по порядку. В связном списке почти каждый адрес, который вы попытаетесь посетить, не будет связан с предыдущим и соответственно не будет находиться в кеше процессора. При использовании вектора (или массива) дело обстоит с точностью до наоборот: каждый посещаемый вами адрес связан с предыдущим простым линейным соотношением, и процессор сможет подготовить значения к моменту, когда они вам понадобятся.

Даже если ваши данные имеют более сложную организацию, чем простой список значений, часто все равно есть возможность использовать вектор для их хранения. Для примера, ближе к концу главы, мы посмотрим, как с помощью вектора можно реализовать стек или приоритетную очередь.

Изменение размера std::vector

`std::vector` обладает целым семейством функций-членов для добавления и удаления элементов. Эти функции-члены отсутствуют в `std::array`, потому что `std::array` не поддерживает возможность изменения размеров; но имеются в большинстве других контейнеров, о которых мы будем говорить в этой главе. Поэтому сейчас самое время познакомиться с ними.

Начнем с двух простейших операций, характерных для типа `vector`: `.resize()` и `.reserve()`.

`vec.reserve(c)` изменяет емкость вектора – она «резервирует» место в памяти для заданного количества `c` элементов. Если `c <= vec.capacity()`, тогда ничего не происходит; но если `c > vec.capacity()`, тогда происходит перераспределение памяти для внутреннего массива. Перераспределение выполняется по алгоритму, эквивалентному следующему:

```
template<typename T>
inline void destroy_n_elements(T *p, size_t n)
{
  for (size_t i = 0; i < n; ++i) {
    p[i].~T();
  }
```

```
}

template<typename T>
class vector {
  T *ptr_ = nullptr;
  size_t size_ = 0;
  size_t capacity_ = 0;

public:
  // ...

  void reserve(size_t c) {
    if (capacity_ >= c) {
      // ничего не делать
      return;
    }

    // Пока просто игнорируем проблему
    // "Что делать, если вызов malloc потерпел неудачу?"
    T *new_ptr = (T *)malloc(c * sizeof (T));

    for (size_t i=0; i < size_; ++i) {
      if constexpr (std::is_nothrow_move_constructible_v<T>) {
        // Если элемент можно переместить, не рискуя получить
        // исключение, тогда переместить его.
        ::new (&new_ptr[i]) T(std::move(ptr_[i]));
      } else {
        // Если перемещение элементов может вызвать исключение,
        // тогда копировать элементы, пока эта операция завершается
        // успехом; затем уничтожить старые элементы.
        try {
          ::new (&new_ptr[i]) T(ptr_[i]);
        } catch (...) {
          destroy_n_elements(new_ptr, i);
          free(new_ptr);
          throw;
        }
      }
    }

    // Если элементы были благополучно перемещены или скопированы,
    // уничтожить старый массив и записать в ptr_ указатель на новый.
    destroy_n_elements(ptr_, size_);
    free(ptr_);
    ptr_ = new_ptr;
    capacity_ = c;
  }

  ~vector() {
    destroy_n_elements(ptr_, size_);
```

```
    free(ptr_);
  }
};
```

Читавшие эту книгу по порядку могли заметить, что критически важный цикл `for` в этой функции `.reserve()` очень напоминает реализацию `std::uninitialized_copy(a,b,c)` из главы 3 «Алгоритмы с парами итераторов». На самом деле в реализации `.reserve()` контейнера, не зависящего от используемого диспетчера памяти (см. главу 8 «Диспетчеры памяти»), можно повторно использовать стандартный алгоритм:

```
// Если элемент можно переместить, не рискуя получить
// исключение, тогда переместить его.
std::conditional_t<
  std::is_nothrow_move_constructible_v<T>,
  std::move_iterator<T*>,
  T*
  > first(ptr_);

try {
  // Переместить или скопировать элементы с помощью стандартного алгоритма.
  std::uninitialized_copy(first, first + size_, new_ptr);
} catch (...) {
  free(new_ptr);
  throw;
}

// Если элементы были благополучно перемещены или скопированы,
// уничтожить старый массив и записать в ptr_ указатель на новый.
std::destroy(ptr_, ptr_ + size_);
free(ptr_);
ptr_ = new_ptr;
capacity_ = c;
```

`vec.resize(s)` изменяет размер вектора – она усекает элементы в конце вектора (попутно вызывая их деструкторы) или добавляет в конец новые элементы (вызывая конструктор по умолчанию), пока размер вектора не станет равным `s`. Если `s > vec.capacity()`, происходит перераспределение внутреннего массива, как при вызове `.reserve()`.

Возможно, также вы заметили, что при перераспределении памяти для массива происходит изменение адресов элементов: адрес `vec[0]` перед перераспределением отличается от адреса `vec[0]` после перераспределения. Любые указатели на старые элементы вектора после этой операции становятся недействительными. А поскольку `std::vector::iterator`, по сути, является простым указателем, любые *итераторы*, ссылающиеся на старые элементы вектора, также становятся недействительными. Это явление называют *аннулированием итераторов* (iterator invalidation), и оно является главным источни-

ком ошибок в коде на C++. Будьте внимательны, когда одновременно работаете с итераторами и изменяете размеры векторов!

Вот несколько классических примеров аннулирования итераторов:

```
std::vector<int> v = {3, 1, 4};

auto iter = v.begin();
v.reserve(6); // iter становится недействительным!

// Может показаться, что этот код произведет результат
// {3, 1, 4, 3, 1, 4}; но если первая вставка
// вызовет перераспределение, тогда следующая вставка
// прочитает мусор из недействительного итератора!
v = std::vector{3, 1, 4};
std::copy(
  v.begin(),
  v.end(),
  std::back_inserter(v)
);
```

Вот еще один случай, известный во многих языках программирования, где стирание элемента из контейнера во время итераций порождает тонкую ошибку:

```
auto end = v.end();
for (auto it = v.begin(); it != end; ++it) {
  if (*it == 4) {
    v.erase(it); // ОШИБКА!
  }
}

// Определение конца вектора вызовом .end() в каждом цикле
// исправляет эту ошибку...
for (auto it = v.begin(); it != v.end(); ) {
  if (*it == 4) {
    it = v.erase(it);
  } else {
    ++it;
  }
}

// ...Но намного эффективнее использовать идиому
// стирания-удаления.
v.erase(
  std::remove_if(v.begin(), v.end(), [](auto&& elt) {
    return elt == 4;
  }),
  v.end()
);
```

Вставка и стирание в std::vector

`vec.push_back(t)` добавляет элемент в конец вектора. Вектор не имеет соответствующей функции-члена `.push_front()` из-за отсутствия эффективного способа добавления элементов в *начало* вектора, как видно из диаграммы в начале этого раздела.

`vec.emplace_back(args...)` – шаблонная функция идеальной передачи с переменным числом аргументов, которая действует подобно `.push_back(t)`, но помещает в конец вектора не копию `t`, а создает объект типа `T`, как если бы был вызван конструктор `T(args...)`.

Обе функции, `push_back` и `emplace_back`, имеют «амортизированную постоянную сложность». Чтобы понять, что это означает, представьте, что случится с простейшим вектором, если сто раз подряд вызвать `v.emplace_back()`. С каждым вызовом размер вектора должен немного увеличиваться; из-за этого происходит перераспределение внутреннего массива и перемещение всех `v.size()` элементов из старого массива в новый. В какой-то момент на копирование данных с места на место будет тратиться больше времени, чем на фактическую «вставку в конец» новых данных! К счастью, реализация `std::vector` предусматривает обходное решение, помогающее избежать этой ловушки. Всякий раз, когда операция, такая как `v.emplace_back()`, вызывает перераспределение памяти, вектор выделяет память не для `capacity() + 1` элементов в новом массиве, а для `k * capacity()` элементов (где `k` равно 2 в libc++ и libstdc++ и примерно равно 1.5 в Visual Studio). То есть даже при том, что с ростом вектора перераспределение памяти обходится все дороже, потребность в перераспределении возникает все реже, и стоимость одного вызова `push_back` в среднем остается постоянной. Этот трюк известен как *геометрическое изменение размера*.

`vec.insert(it, t)` вставляет элемент в середину вектора, в позицию, на которую ссылается итератор `it`. Если `it == vec.end()`, вызов `insert` становится эквивалентным вызову `push_back`; если `it == vec.begin()`, этот вызов можно считать неэффективной версией `push_front`. Обратите внимание, что если вставка выполняется в любую позицию, кроме конца вектора, все элементы, следующие во внутреннем массиве за позицией вставки, будут перемещены вправо, чтобы освободить место; эта операция может стоить очень дорого.

Существует несколько перегруженных версий `.insert()`. Вообще говоря, желательно избегать использования любой из них, но вы должны знать их, чтобы расшифровывать некоторые загадочные сообщения об ошибках (или загадочные ошибки времени выполнения), которые появляются, если по ошибке передать `.insert()` неверные аргументы, а механизм перегрузки ошибочно выберет не ту версию, которую вы ожидали:

```
std::vector<int> v = {1, 2};
std::vector<int> w = {5, 6};

// Вставить единственный элемент.
```

```
v.insert(v.begin() + 1, 3);
assert((v == std::vector{1, 3, 2}));

// Вставить n копий единственного элемента.
v.insert(v.end() - 1, 3, 4);
assert((v == std::vector{1, 3, 4, 4, 4, 2}));

// Вставить целый диапазон элементов.
v.insert(v.begin() + 3, w.begin(), w.end());
assert((v == std::vector{1, 3, 4, 5, 6, 4, 4, 2}));

// Вставить список элементов.
v.insert(v.begin(), {7, 8});
assert((v == std::vector{7, 8, 1, 3, 4, 5, 6, 4, 4, 2}));
```

`vec.emplace(it, args...)` относится к `insert`, так же как `emplace_back` относится к `push_back`: это версия идеальной передачи функции из стандарта C++03. Используйте `emplace` и `emplace_back` вместо `insert` и `push_back` везде, где возможно.

`vec.erase(it)` стирает единственный элемент в середине вектора, находящийся в позиции, определяемой итератором `it`. Существует также версия с двумя итераторами, `vec.erase(it1, it2)`, которая стирает непрерывный диапазон элементов. Обратите внимание, это та самая версия с двумя итераторами, которая использовалась в реализации *идиомы стирания-удаления* в предыдущей главе.

Чтобы удалить только последний элемент из вектора, можно использовать вызов `vec.erase(vec.end()-1)` или `vec.erase(vec.end()-1, vec.end())`; но, так как это довольно распространенная операция, в стандартную библиотеку был добавлен синоним `vec.pop_back()`. Вы можете реализовать стек, динамически изменяющий размеры, не используя никаких других функций, кроме `push_back()` и `pop_back()` вектора `std::vector`.

Ловушки vector<bool>

Шаблон `std::vector` имеет один специальный случай: `std::vector<bool>`. Так как тип данных `bool` может иметь только два значения, восемь значений `bool` можно упаковать в один байт. `std::vector<bool>` использует эту оптимизацию, а это значит, что такой вектор занимает памяти в восемь раз меньше, чем можно было бы ожидать.

Недостаток такой упаковки в том, что значение, возвращаемое `vector<bool>::operator[]`, не может иметь тип `bool&`, потому что вектор не хранит логические значения в отдельно адресуемых ячейках памяти. По этой причине `operator[]` возвращает значение специализированного типа `std::vector<bool>::reference`, который преобразуется в тип `bool`, но само таковым не является (такие типы, как в этом случае, часто называют «прокситипами» или «прокси-ссылками»).

```
std::vector<bool> v {
  true, false, true, false, false, false, true, false,
  true, true,
};
v.reserve(60);
```

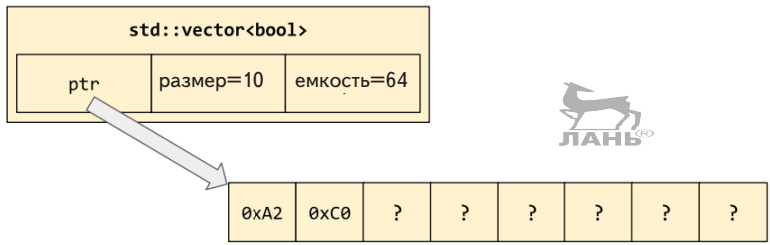


Рис. 4.3. Вектор типа std::vector<bool>

Результат `operator[] const` «официально» имеет тип `bool`, но на практике `operator[] const` в некоторых библиотеках (например, libc++) возвращает прокси-тип. Это означает, что код, использующий `vector<bool>`, не только обладает тонкими особенностями, но иногда оказывается непереносимым; я советую по возможности не использовать `vector<bool>`:

```
std::vector<bool> vb = {true, false, true, false};

// vector<bool>::reference имеет только одну общедоступную функцию-член:
vb[3].flip();
assert(vb[3] == true);

// Следующая строка не компилируется!
// bool& oops = vb[0];

auto ref = vb[0];
assert((!std::is_same_v<decltype(ref), bool>));
assert(sizeof vb[0] > sizeof (bool));

if (sizeof std::as_const(vb)[0] == sizeof (bool)) {
  puts("Your library vendor is libstdc++ or Visual Studio");
} else {
  puts("Your library vendor is libc++");
}
```

Ловушки в конструкторах перемещения без noexcept

Вспомним реализацию `vector::resize()` из раздела «Изменение размера std::vector». Когда изменяется размер вектора, происходит перераспределение его внутреннего массива и перемещение элементов в новый массив – если только тип элемента поддерживает «конструирование без исключений при перемещении», иначе элементы копируются! То есть изменение размера

вектора вашего собственного типа будет необоснованно «пессимизировано»[1], если только вы не снабдите свой конструктор перемещения спецификацией `noexcept`.

Взгляните на определения следующих классов:

```
struct Bad {
  int x = 0;
  Bad() = default;
  Bad(const Bad&) { puts("copy Bad"); }
  Bad(Bad&&) { puts("move Bad"); }
};

struct Good {
  int x = 0;
  Good() = default;
  Good(const Good&) { puts("copy Good"); }
  Good(Good&&) noexcept { puts("move Good"); }
};

class ImplicitlyGood {
  std::string x;
  Good y;
};

class ImplicitlyBad {
  std::string x;
  Bad y;
};
```

Проверим поведение этих классов по отдельности, как показано ниже. Вызов `test()` выведет `"copy Bad move Good copy Bad move Good"`. Похоже на мантру!

```
template<class T>
void test_resizing()
{
  std::vector<T> vec(1);
  // Вынудить вектор выполнить перераспределение внутреннего массива.
  vec.resize(vec.capacity() + 1);
}

void test()
{
  test_resizing<Good>();
  test_resizing<Bad>();
  test_resizing<ImplicitlyGood>();
  test_resizing<ImplicitlyBad>();
}
```

Это очень тонкая и малопонятная особенность, которая способна существенно влиять на производительность вашего кода на C++. Главное правило:

[1] Пессимизация – противоположность оптимизации. – *Прим. перев.*

всякий раз, объявляя свой конструктор перемещения или функцию перестановки (swap), добавляйте спецификатор noexcept.

Быстрый гибрид: std::deque<T>

По аналогии с std::vector, двусторонняя очередь std::deque служит интерфейсом к непрерывному массиву – она поддерживает произвольный доступ и хранит элементы в непрерывных блоках памяти, что обеспечивает дружественность к кешу процессора. Но, в отличие от vector, std::deque поддерживает «фрагментарную» непрерывность хранения элементов. Единственный контейнер deque может состоять из произвольного количества «фрагментов», каждый из которых хранит фиксированное число элементов. Вставка новых элементов с любого конца контейнера стоит достаточно дешево; но вставка в середину все еще обходится дорого. В памяти deque хранится примерно так, как показано на рис. 4.4.

```
std::deque<char> dq {42, 43, 44};
```

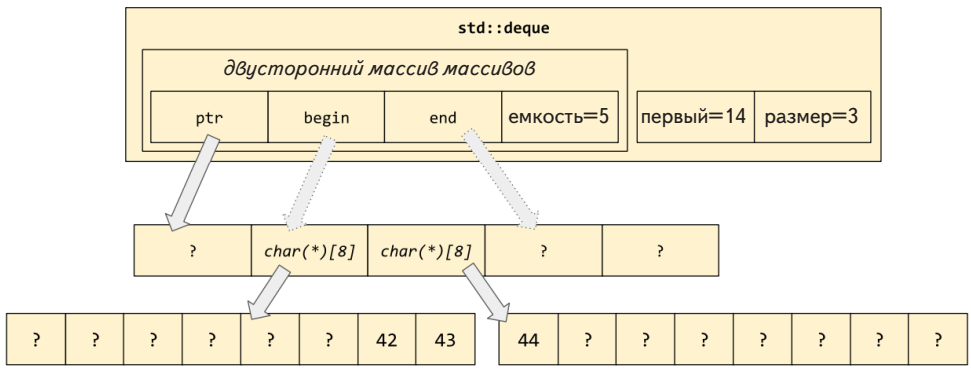


Рис. 4.4. Контейнер deque

std::deque<T> поддерживает те же функции-члены, что и std::vector<T>, включая перегруженный operator[]. В дополнение к векторным методам push_back и pop_back, deque предоставляет эффективные версии push_front и pop_front.

Обратите внимание, что многократный вызов push_back для добавления элементов в вектор рано или поздно приведет к перераспределению внутреннего массива и аннулированию всех ваших итераторов, указателей и ссылок на элементы внутри контейнера. При использовании deque также происходит аннулирование итераторов, но отдельные элементы никогда не меняют своих адресов, если только вы не будете вставлять или стирать элементы в середине очереди (в этом случае элементы с одного или с другого конца будут сдвинуты внутрь, чтобы заполнить пробел):

```
std::vector<int> vec = {1, 2, 3, 4};
std::deque<int> deq = {1, 2, 3, 4};
int *vec_p = &vec[2];
int *deq_p = &deq[2];
for (int i=0; i < 1000; ++i) {
  vec.push_back(i);
  deq.push_back(i);
}
assert(vec_p != &vec[2]);
assert(deq_p == &deq[2]);
```

Другое достоинство `std::deque<T>` – отсутствие специального случая для `std::deque<bool>`; контейнер обеспечивает единообразный общедоступный интерфейс независимо от типа `T`.

Недостаток `std::deque<T>` – операции инкремента, декремента и разыменования итераторов обходятся намного дороже, потому что им приходится шагать по массиву указателей, как показано на рис. 4.4. Это достаточно серьезный недостаток, чтобы отдать предпочтение вектору, если не требуется быстрая вставка и удаление с обоих концов контейнера.

Особый набор возможностей: std::list<T>

Контейнер `std::list<T>` представляет связный список элементов в памяти. Схематически его можно представить, как показано на рис. 4.5.

```
std::list<char> lst {42, 43, 44};
```

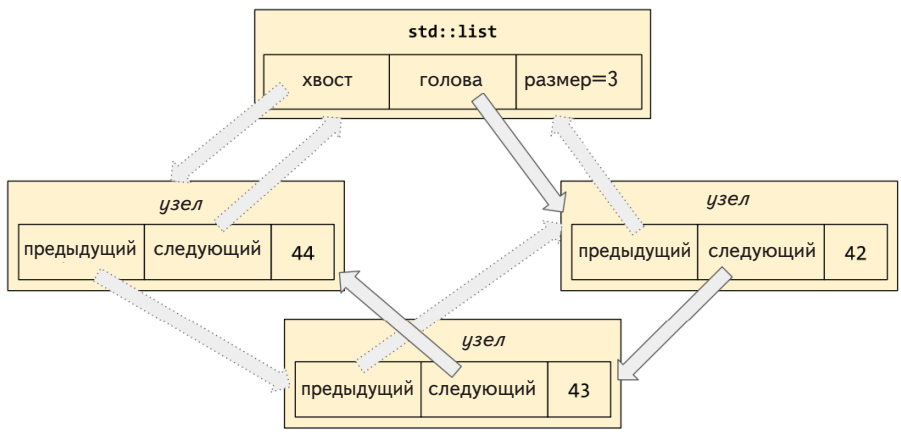


Рис. 4.5. Связный список std::list<T>

Обратите внимание, что каждый узел в списке содержит указатели на следующий и предыдущий узлы, то есть это двусвязный список. Преимущество двусвязного списка – в возможности перемещения итераторов в обоих на-

правлениях – вперед и назад. То есть итератор `std::list<T>::iterator` является *двунаправленным* (но не с произвольным доступом; для получения *n*-го элемента списка все еще требуется время O(*n*)).

`std::list` поддерживает почти те же операции, что и `std::vector`, кроме тех, которые требуют произвольного доступа (как `operator[]`). И он может позволить себе функции-члены вставки и извлечения элементов из начала и конца списка, потому что для их реализации не требуется использовать дорогостоящие операции перемещения.

В целом `std::list` имеет намного худшую производительность, чем структуры с непрерывными массивами, такие как `std::vector` или `std::deque`, потому что следование по указателям, ссылающимся на адреса, разбросанные в «случайном» порядке, для кеша процессора намного сложнее, чем следование по указателям, ссылающимся на адреса, идущим подряд. Поэтому `std::list` обычно следует рассматривать как нежелательный контейнер; используйте его только в алгоритмах, где его преимущество перед `vector` неоспоримо.

Какие отличительные особенности имеет std::list?

Во-первых, списки *не страдают проблемой аннулирования итераторов*! `lst.push_back(v)` и `lst.push_front(v)` всегда выполняются за постоянное время и не требуют «перераспределения» или «перемещения» данных.

Во-вторых, многие операции, меняющие содержимое контейнера, которые дорого стоят при использовании вектора или требуют создания временного хранилища, для списков обходятся очень дешево. Вот несколько примеров:

`lst.splice(it, otherlst)` добавляет содержимое `otherlst` в `lst`, как при многократном вызове `lst.insert(it++, other_elt)`; за исключением того, что «вставленные» узлы фактически исключаются из списка `otherlst`. Операция `splice` целиком может быть выполнена перестановкой пары указателей. После этой операции `otherlst.size() == 0`.

`lst.merge(otherlst)` аналогично добавляет содержимое `otherlst` в `lst` простой перестановкой указателей, но имеет эффект «слияния сортированных списков». Например:

```
std::list<int> a = {3, 6};
std::list<int> b = {1, 2, 3, 5, 6};

a.merge(b);
assert(b.empty());
assert((a == std::list{1, 2, 3, 3, 5, 6, 6}));
```

Как обычно для операций в STL, вовлекающих сравнение, существует версия, принимающая компаратор: `lst.merge(otherlst, less)`.

Другая операция, которую можно выполнить простой перестановкой указателей, – обращение списка на месте:

`lst.reverse()` меняет местами все указатели «следующий» и «предыдущий» так, что голова становится хвостом списка, и наоборот.

Обратите внимание, что все эти операции *изменяют список на месте* и обычно возвращают `void`.

Еще одна операция, которая стоит недорого в связных списках (но не в непрерывных контейнерах), – удаление элементов. Как рассказывалось в главе 3 «Алгоритмы с парами итераторов», стандартная библиотека STL включает такие алгоритмы, как `std::remove_if` и `std::unique` для использования с непрерывными контейнерами; эти алгоритмы перемещают «удаляемые» элементы в конец контейнера, чтобы потом их можно было удалить все разом одним вызовом `erase()`. В `std::list` перемещение элементов обходится дороже, чем непосредственное удаление на месте. Поэтому `std::list` включает следующие функции-члены с именами, к сожалению, похожими на имена нестирающих алгоритмов:

- `lst.remove(v)` удаляет и стирает все элементы, равные `v`;
- `lst.remove_if(p)` удаляет и стирает все элементы `e`, удовлетворяющие предикату `p(e)` с одним аргументом;
- `lst.unique()` удаляет и стирает все элементы, кроме первого в каждой группе идущих подряд уникальных элементов. Как обычно для операций, использующих сравнение, существует версия с компаратором: `lst.unique(p)`, удаляющая и стирающая элементы `e2`, для которых `p(e1, e2)` возвращает `true`;
- `lst.sort()` сортирует список на месте. Это особенно полезная операция, потому что алгоритм `std::sort(ctr.begin(), ctr.end())` не работает с `std::list::iterator`, не поддерживающим произвольный доступ.

Довольно странным выглядит то обстоятельство, что `lst.sort()` может сортировать содержимое только всего контейнера, а не поддиапазон, как это делает `std::sort`. Если вам потребуется отсортировать только часть списка `lst`, это можно сделать – как показано ниже – простой перестановкой пары указателей!

```
std::list<int> lst = {3, 1, 4, 1, 5, 9, 2, 6, 5};
auto begin = std::next(lst.begin(), 2);
auto end = std::next(lst.end(), -2);

// Сортировать диапазон [begin, end)
std::list<int> sub;
sub.splice(sub.begin(), lst, begin, end);
sub.sort();
lst.splice(end, sub);
assert(sub.empty());

assert((lst == std::list{3, 1, 1, 2, 4, 5, 9, 6, 5}));
```

Список без удобств std::forward_list<T>

Стандартный контейнер `std::forward_list<T>` – это связный список, подобный `std::list`, но с меньшим количеством удобств – он не позволяет получить его размер и выполнить обход элементов в обратном направлении. В памяти он хранится так же, как `std::list<T>`, но его узлы имеют меньший размер (см. рис. 4.6).

```
std::forward_list<char> lst {42, 43, 44};
```

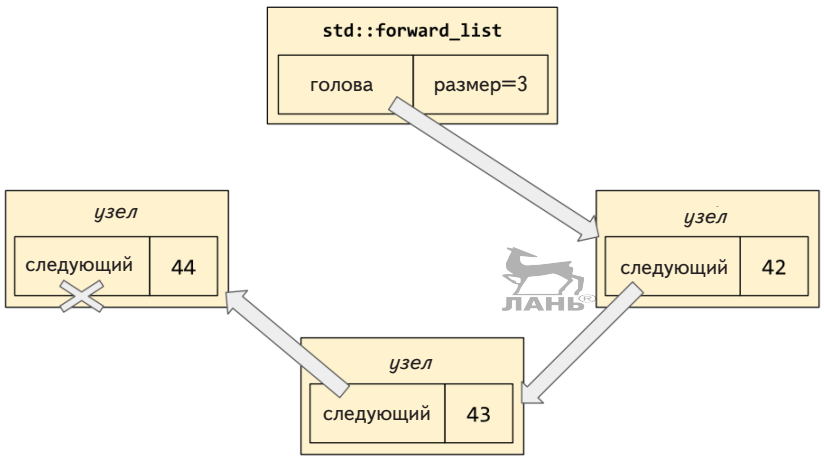


Рис. 4.6. Упрощенный список std::forward_list<T>

Тем не менее `std::forward_list` сохраняет почти «отличительные особенности» `std::list`. Единственные операции, которых он не поддерживает: `splice` (потому что она выполняет операцию вставки «перед» заданным итератором) и `push_back` (потому что ей требуется выполнять поиск хвоста за постоянное время).

`forward_list` заменяет эти отсутствующие функции-члены их версиями `_after`:

- `flst.erase_after(it)` стирает элемент, *следующий за* указанным;
- `flst.insert_after(it, v)` вставляет новый элемент *вслед за* указанным;
- `flst.splice_after(it, otherflst)` вставляет элементы из `otherflst` *вслед за* указанным.

Так же как в случае с `std::list`, используйте `forward_list` только в алгоритмах, где его преимущество неоспоримо.

Абстракции с использованием std::stack<T> и std::queue<T>

Мы познакомились уже с тремя стандартными контейнерами, имеющим функции-члены `push_back()` и `pop_back()` (а также `back()`, которая не упоминалась выше – она возвращает ссылку на последний элемент в контейнере). Эти операции понадобятся нам, если у нас появится желание реализовать структуру данных типа стек.

Стандартная библиотека включает удобную абстракцию стека с именем (что совершенно неудивительно) `std::stack`. Однако, в отличие от контейнеров, представленных до сих пор, `std::stack` принимает дополнительный параметр типа.

`std::stack<T, Ctr>` представляет стек с элементами типа `T`, внутреннее хранилище которого управляется экземпляром контейнерного типа `Ctr`. Например, `stack<T, vector<T>>` использует вектор для управления элементами; `stack<T, list<T>>` использует список и т. д. По умолчанию шаблонный параметр `Ctr` принимает значение `std::deque<T>`; как вы помните, `deque` потребляет больше памяти, чем вектор, но обладает тем преимуществом, что никогда не производит перераспределение внутреннего массива и не перемещает элементы после вставки.

Работая с `std::stack<T, Ctr>`, вы должны ограничивать себя только операциями `push` (соответствует операции `push_back` внутреннего контейнера), `pop` (соответствует операции `pop_back`), `top` (соответствует операции `back`) и некоторыми другими вспомогательными операциями, такими как `size` и `empty`:

```
std::stack<int> stk;
stk.push(3); stk.push(1); stk.push(4);
assert(stk.top() == 4);
stk.pop();
assert(stk.top() == 1);
stk.pop();
assert(stk.top() == 3);
```

Одной из необычных особенностей `std::stack` является поддержка операторов сравнения `==`, `!=`, `<`, `<=`, `>` и `>=`, которые сравнивают внутренние контейнеры (используя семантику, определяемую самими контейнерами). Поскольку обычно внутренние контейнеры сравнивают свои значения в лексикографическом порядке, два стека также будут сравниваться в «лексикографическом порядке, снизу вверх».

```
std::stack<int> a, b;
a.push(3); a.push(1); a.push(4);
b.push(2); b.push(7);

assert(a != b);
assert(a.top() < b.top()); // то есть 4 < 7
assert(a > b);             // потому что 3 > 2
```

На практике, безусловно, найдут применение == и !=, также можно использовать operator<, чтобы получить единообразное упорядочение для std::set или std::map; но в первый момент все это выглядит необычно!

В стандартной библиотеке имеется также абстракция «очереди». std::queue поддерживает методы push_back и pop_front (соответствующие методам push_back и pop_front внутреннего контейнера), а еще несколько других вспомогательных методов, таких как front, back, size и empty.

Понимая, что контейнер должен обеспечивать максимальную эффективность этих операций, вы без труда угадаете значение *по умолчанию* для Ctr. Да, это std::deque<T>, двусторонняя очередь с низкими накладными расходами.

Обратите внимание, что если бы вы решили реализовать очередь «с нуля» на основе std::deque<T>, у вас на выбор было бы два варианта: вставлять элементы в начало deque и извлекать с конца или вставлять в конец и извлекать из начала. Для стандартной реализации std::queue<T, std::deque<T>> выбран второй вариант – вставка в конец и извлечение из начала. Это легко запоминается по аналогии с обычными очередями в реальном мире. Вставая в очередь в кассу кинотеатра или в столовой, вы присоединяетесь к ней с конца, и вас обслуживают, когда вы оказываетесь в начале, и никогда наоборот! Всегда полезно выбирать технические термины (такие как queue, front и back), технический смысл которых точно отражает аналоги из реального мира.

Удобный адаптер: std::priority_queue<T>

В главе 3 «Алгоритмы с парами итераторов» мы познакомились с семейством алгоритмов «кучи»: make_heap, push_heap и pop_heap. Эти алгоритмы можно использовать, чтобы придать диапазону элементов свойство невозрастания пирамиды (max-heap property). Если поддерживать для данных свойство невозрастания пирамиды на постоянной основе, получится структура, широко известная как *приоритетная очередь*. В книгах, посвященных программированию, приоритетная очередь часто описывается как разновидность бинарного дерева, но, как мы видели в главе 3 «Алгоритмы с парами итераторов», в свойстве невозрастания пирамиды нет ничего, что требовало бы использования древовидной структуры.

Стандартный контейнер std::priority_queue<T, Ctr, Cmp> – это приоритетная очередь, внутренне представленная экземпляром контейнера Ctr, для элементов которого непрерывно поддерживается свойство невозрастания пирамиды (как определяет экземпляр типа компаратора Cmp).

По умолчанию Ctr получает значение std::vector<T>. Напомню, что вектор является самым эффективным контейнером; единственная причина, почему для абстракций std::stack и std::queue по умолчанию выбирается deque, состоит в том, что они перемещают элементы после их вставки. Но в приоритетной очереди элементы перемещаются постоянно, вверх и вниз, для поддержания свойства невозрастания пирамиды, когда происходит вставка

или удаление других элементов. То есть использование deque в качестве внутреннего контейнера не дает никаких преимуществ; поэтому стандартная библиотека следует тому же правилу, которое я не устаю повторять: используйте std::vector, если у вас нет особых причин применять что-то другое!

Для параметра Cmp стандартная библиотека по умолчанию использует тип std::less<T>, который представляет operator<. Иными словами, контейнер std::priority_queue по умолчанию применяет тот же компаратор, что и алгоритмы std::push_heap и std::pop_heap из главы 3 «Алгоритмы с парами итераторов».

Основными функциями-членами контейнера std::priority_queue<T, Ctr> являются push, pop и top. Концептуально элемент в начале внутреннего контейнера находится на «вершине» (top) кучи. Важно помнить, что в контейнерах с поддержкой свойства невозрастания пирамиды элемент на «вершине» кучи является *наибольшим* – как в карточной игре «King of the Hill», где побеждает карта наибольшего достоинства и оказывается на вершине колоды.

- pq.push(v) вставляет новый элемент в приоритетную очередь, действуя подобно std::push_heap() внутреннего контейнера;
- pq.top() возвращает ссылку на элемент, в данный момент находящийся на вершине приоритетной очереди, действуя подобно ctr.front() внутреннего контейнера;
- pq.pop() выталкивает максимальный элемент из очереди и обновляет кучу, действуя подобно std::pop_heap() внутреннего контейнера.

Чтобы получить контейнер со свойством неубывания пирамиды, достаточно просто поменять компаратор, передаваемый в шаблон priority_queue:

```
std::priority_queue<int> pq1;
std::priority_queue<int, std::vector<int>, std::greater<>> pq2;

for (int v : {3, 1, 4, 1, 5, 9}) {
  pq1.push(v);
  pq2.push(v);
}

assert(pq1.top() == 9); // max-heap by default
assert(pq2.top() == 1); // min-heap by choice
```

Деревья: std::set<T> и std::map<K, V>

Шаблонный класс std::set<T> реализует интерфейс «множества уникальных элементов» любого типа T, реализующего operator<. Как это обычно для операций в STL, вовлекающих сравнение, существует версия, принимающая компаратор: std::set<T, Cmp>, обеспечивающая «уникальность» с использованием Cmp(a,b) вместо (a < b) для сортировки элементов.

Концептуально `std::set` – это бинарное дерево поиска, подобное `TreeSet` в Java. Во всех популярных реализациях используется *красно-черное* дерево, которое является особой разновидностью самобалансирующегося бинарного дерева поиска: даже если в такое дерево постоянно вставлять и удалять элементы, оно никогда не окажется слишком несбалансированным, а это значит, что в среднем вставка и поиск всегда будут выполняться за время O(log *n*). Обратите внимание на количество указателей, используемое в структуре памяти.

```
std::set<char> s {42, 43, 44};
```

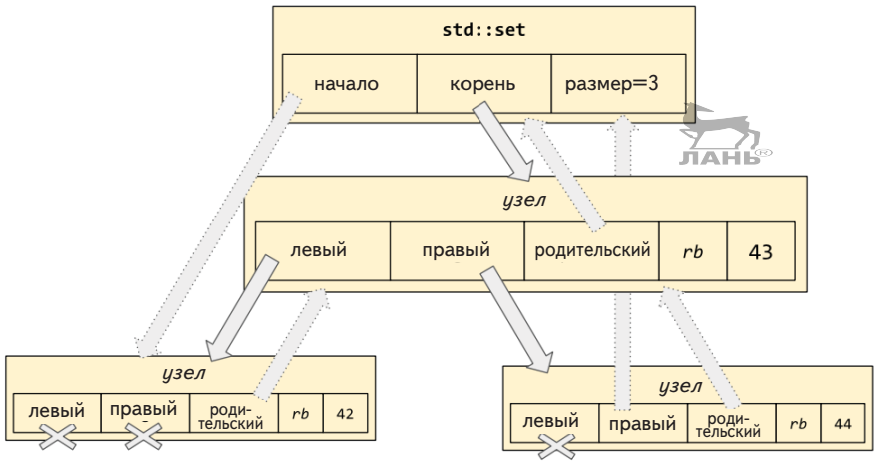


Рис. 4.7. Организация контейнера std::set в памяти

Так как по определению элементы бинарного дерева поиска хранятся в порядке сортировки (от меньших к большим), функции-члены `push_front` или `push_back` не имеют смысла для `std::set`. Вместо этого для добавления элемента `v` во множество используется `s.insert(v)`; а для удаления – `s.erase(v)` или `s.erase(it)`:

```
std::set<int> s;
for (int i : {3, 1, 4, 1, 5}) {
  s.insert(i);
}

// Элементы во множестве отсортированы и не повторяются.
assert((s == std::set{1, 3, 4, 5}));

auto it = s.begin();
assert(*it == 1);
s.erase(4);
s.erase(it); // стереть *it - элемент 1

assert((s == std::set{3, 5}));
```

Возвращаемым значением `s.insert(v)` является результат вставки. Когда вставка выполняется в вектор, возможны только два результата: значение, успешно вставленное в вектор (и мы получаем обратно итератор на вновь вставленный элемент), или исключение, если вставка не увенчалась успехом. Когда вставка выполняется во множество, добавляется третий возможный результат: вставка может оказаться неудачной, потому что во множестве уже имеется копия `v`! Это не та «ошибка», требующая исключительной обработки, но все же достаточно важное обстоятельство, о котором необходимо уведомить вызывающий код. Поэтому `s.insert(v)` всегда возвращает пару значений: `ret.first` – обычный итератор, указывающий на копию `v` в структуре данных (не важно, когда она была вставлена, только что или раньше), и `ret.second`, содержащее `true`, если элемент `v` вставлен только что, и `false`, если элемент `v` был вставлен раньше:

```
std::set<int> s;
auto [it1, b1] = s.insert(1);
assert(*it1 == 1 && b1 == true);

auto [it2, b2] = s.insert(2);
assert(*it2 == 2 && b2 == true);

auto [it3, b3] = s.insert(1); // еще раз
assert(*it3 == 1 && b3 == false);
```

Квадратные скобки в объявлениях переменных в предыдущем фрагменте – это структурированная привязка, появившаяся в C++17.

Как показывает предыдущий пример, элементы множества хранятся упорядоченными – не только концептуально, но и визуально, то есть `*s.begin()` вернет наименьший элемент, а `*std::prev(s.end())` – наибольший. Итерации по множеству с применением стандартного алгоритма или цикла `for` с диапазоном дадут вам элементы в порядке возрастания (напомню, что понятие «возрастание» определяется компаратором – параметром `Cmp` в шаблоне класса `set`).

Древовидная структура множества предполагает, что некоторые стандартные алгоритмы, такие как `std::find` и `std::lower_bound` (глава 3 «Алгоритмы с парами итераторов»), смогут работать со множествами, но неэффективно – итераторы алгоритма будут тратить массу времени на восхождение и спуск в дереве, тогда как при наличии доступа к самой древовидной структуре мы могли бы сами спуститься вниз от корня дерева и быстро найти позицию элемента. По этой причине `std::set` реализует функции-члены, которые можно использовать взамен неэффективных алгоритмов:

- вместо `std::find(s.begin(), s.end(), v)` используйте `s.find(v)`;
- вместо `std::lower_bound(s.begin(), s.end(), v)` используйте `s.lower_bound(v)`;
- вместо `std::upper_bound(s.begin(), s.end(), v)` используйте `s.upper_bound(v)`;
- вместо `std::count(s.begin(), s.end(), v)` используйте `s.count(v)`;
- вместо `std::equal_range(s.begin(), s.end(), v)` используйте `s.equal_range(v)`.

Обратите внимание, что `s.count(v)` может вернуть только 0 или 1, потому что элементы множества уникальны. Это обстоятельство превращает `s.count(v)` в удобный синоним для операции проверки членства во множестве – которая на языке Python записывается как `v in s`, а на языке Java как `s.contains(v)`.

`std::map<K, V>` (словарь) имеет много общего с `std::set<K>`, с той лишь разницей, что каждому ключу K можно поставить в соответствие значение V; как результат эта структура данных оказывается очень похожей на `TreeMap` в Java или `dict` в Python. Как обычно, существует также версия `std::map<K, V, Cmp>`, которую можно использовать, если понадобится задать иной порядок сортировки ключей, отличный от естественного `K::operator<`. Вам нечасто будет приходить в голову мысль, что `std::map` – это «всего лишь тонкая обертка вокруг множества `std::set` пар», но посмотрите, как хранится `std::map` в памяти.

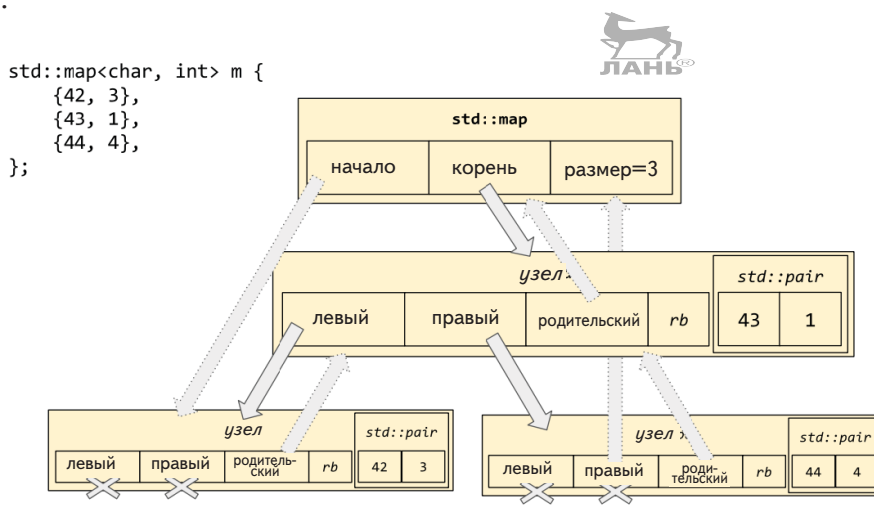


Рис. 4.8. Организация контейнера std::map в памяти

`std::map` поддерживает индексирование с помощью `operator[]`, но с одной необычной особенностью. Поведение инструкции индексирования, например `vec[42]`, не определено для вектора нулевого размера. Но при попытке

индексировать словарь нулевого размера инструкцией `m[42]`, в словарь будет вставлена пара ключ/значение `{42, {}}`, и вызывающий код получит ссылку на второй элемент этой пары!

Такое необычное поведение фактически помогает писать простой и понятный код:

```
std::map<std::string, std::string> m;
m["hello"] = "world";
m["quick"] = "brown";
m["hello"] = "dolly";
assert(m.size() == 2);
```

Но может приводить к путанице, если ослабить внимание:

```
assert(m["literally"] == "");
assert(m.size() == 3);
```

Обратите внимание, что словари не имеют `operator[] const`, потому что `operator[]` всегда оставляет возможность вставить новую пару ключ/значение в `*this`. Если вы объявили константный словарь или простой словарь, но не хотите вставлять новые элементы в него, тогда используйте для проверки наличия ключа `m.find(k)`. Другая причина избегать `operator[]`: если тип `V` не имеет значения, конструируемого по умолчанию, тогда `operator[]` просто не скомпилируется. В таком случае (а по большому счету в *любом случае*) используйте `m.insert(k, v)` или `m.emplace(k, v)` для вставки новой пары ключ/значение в требуемом виде вместо создания значения по умолчанию только ради того, чтобы тут же заменить значение по умолчанию вновь созданным. Например:

```
// Как ни странно, но под "value_type" подразумевается тип всей пары ключ/значение.
// Типы K и V называются "key_type" и "mapped_type" соответственно.
using Pair = decltype(m)::value_type;

if (m.find("hello") == m.end()) {
  m.insert(Pair{"hello", "dolly"});

  // ...или эквивалент...
  m.emplace("hello", "dolly");
}
```

Мудрость, накопленная в пост-C++11 мире, гласит, что `std::map` и `std::set`, будучи основанными на деревьях указателей, настолько недружественные к кешу процессора, что их всегда желательно избегать и взамен использовать `std::unordered_map` и `std::unordered_set`.

Замечание о прозрачных компараторах

В последнем примере я написал `m.find("hello")`. Обратите внимание, что аргумент `"hello"` имеет тип `const char[6]`, тогда как `decltype(m)::key_type` – это `std::string` и (поскольку мы не определили ничего специального)

decltype(m)::key_compare — это std::less<std::string>. То есть, вызывая m.find("hello"), мы вызываем функцию, первый параметр которой имеет тип std::string, а значит, неявно вызываем конструктор std::string("hello"), чтобы передать аргумент в find. В общем случае аргумент m.find неявно преобразуется в тип decltype(m)::key_type, что может обходиться довольно дорого.

При правильной работе operator< можно избежать этих накладных расходов, заменив компаратор m некоторым классом с гетерогенным operator(), который также определяет член typedef is_transparent, например:

```
struct MagicLess {
  using is_transparent = std::true_type;

  template<class T, class U>
  bool operator()(T&& t, U&& u) const {
    return std::forward<T>(t) < std::forward<U>(u);
  }
};

void test()
{
  std::map<std::string, std::string, MagicLess> m;

  // В STL имеется std::less<> -- синоним для MagicLess.
  std::map<std::string, std::string, std::less<>> m2;

  // Теперь 'find' не будет конструировать std::string!
  auto it = m2.find("hello");
}
```

Здесь творится настоящее «волшебство» внутри библиотечной реализации std::map; функция-член find проверяет наличие члена is_transparent и в соответствии с результатом проверки меняет свое поведение. Функции-члены count, lower_bound, upper_bound и equal_range также меняют свое поведение. Но, по странной прихоти, функция-член erase не делает этого! Вероятно, это связано с тем, что для механизма перегрузки было бы слишком сложно отличать преднамеренный вызов m.erase(v) от преднамеренного вызова m.erase(it). Но, как бы то ни было, при желании можно организовать гетерогенное сравнение в операции стирания:

```
auto [begin, end] = m.equal_range("hello");
m.erase(begin, end);
```

Необычные std::multiset<T> и std::multimap<K, V>

На языке STL «множество» – это упорядоченная коллекция неповторяющихся элементов. Соответственно, «мультимножество» (multiset) – это упорядочен-

ная коллекция возможно повторяющихся элементов! В памяти мультимножество хранится точно так же, как `std::set`. Обратите внимание на рис. 4.9, что `std::multiset` допускает хранение двух элементов 42.

```
std::multiset<char> s {42, 42, 44};
```

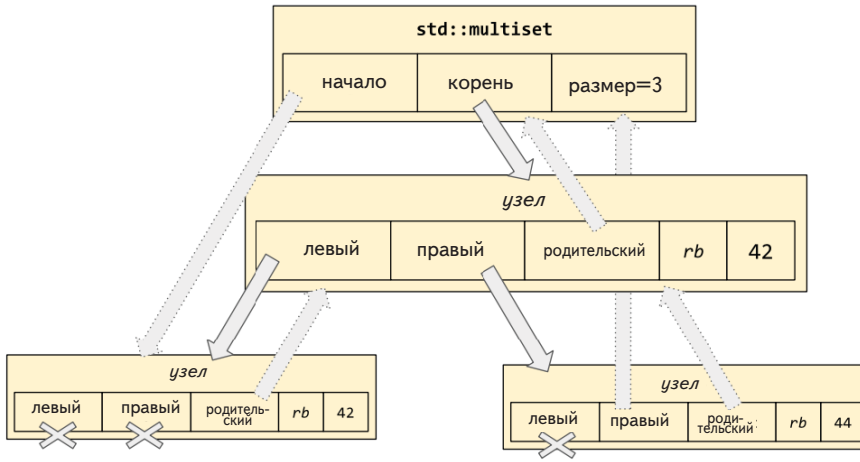


Рис. 4.9. Организация контейнера std::multiset в памяти

`std::multiset<T, Cmp>` действует точно так же, как `std::set<T, Cmp>`, но, в отличие от последнего, может хранить повторяющиеся элементы. То же относится к `std::multimap<K, V, Cmp>`:

```
std::multimap<std::string, std::string> mm;
mm.emplace("hello", "world");
mm.emplace("quick", "brown");
mm.emplace("hello", "dolly");
assert(mm.size() == 3);

// Пары ключ/значение хранятся в порядке сортировки.
// Пары с одинаковыми ключами гарантированно хранятся
// в порядке вставки.
auto it = mm.begin();
using Pair = decltype(mm)::value_type;
assert(*(it++) == Pair("hello", "world"));
assert(*(it++) == Pair("hello", "dolly"));
assert(*(it++) == Pair("quick", "brown"));
```

Метод `mm.find(v)` мультимножества и мультисловаря возвращает итератор, ссылающийся на *некоторый* элемент (или пару ключ/значение), соответствующий `v` – необязательно на первый в порядке итераций. `mm.erase(v)` стирает все элементы (или пары ключ/значение) с ключами, равными `v`. А поддержка индексации `mm[v]` вообще отсутствует. Например:

```
std::multimap<std::string, std::string> mm = {
  {"hello", "world"},
  {"quick", "brown"},
  {"hello", "dolly"},
};
assert(mm.count("hello") == 2);
mm.erase("hello");
assert(mm.count("hello") == 0);
```

Перемещение элементов без перемещения

Напомню, что `std::list` позволяет объединять списки, перемещать элементы из одного списка в другой и т. д., используя «отличительные особенности» типа `std::list`. Контейнеры с древовидной организацией, начиная со стандарта C++17, тоже приобрели аналогичные особенности!

Синтаксис объединения двух множеств или словарей (или мультимножеств или мультисловарей) обманчиво похож на синтаксис объединения сортированных списков `std::list`:

```
std::map<std::string, std::string> m = {
  {"hello", "world"},
  {"quick", "brown"},
};

std::map<std::string, std::string> otherm = {
  {"hello", "dolly"},
  {"sad", "clown"},
};

// Выглядит очень знакомо!
m.merge(otherm);

assert((otherm == decltype(m){
  {"hello", "dolly"},
}));

assert((m == decltype(m){
  {"hello", "world"},
  {"quick", "brown"},
  {"sad", "clown"},
}));
```

Но обратите внимание, что происходит при наличии одинаковых элементов! Одинаковые элементы не перемещаются; они остаются в словаре справа! Это в корне не соответствует ожиданиям, если, к примеру, вы пришли из языка Python, где `d.update(otherd)` вставляет все элементы из правого словаря в левый, перезаписывая элементы, если они уже существуют.

Эквивалент `d.update(otherd)` в C++ – это `m.insert(otherm.begin(), otherm.end())`. Единственный случай, когда имеет смысл использовать `m.merge(otherm)`, если известно, что вы не должны перезаписывать повторя-

ющиеся элементы *и* собираетесь просто стереть значения, оставшиеся в `otherm` (например, если это временный словарь, существующий в ограниченной области видимости).

Другой способ перемещения элементов между контейнерами, основанными на древовидных структурах, – использовать функции-члены `extract` и `insert`:

```
std::map<std::string, std::string> m = {
  {"hello", "world"},
  {"quick", "brown"},
};

std::map<std::string, std::string> otherm = {
  {"hello", "dolly"},
  {"sad", "clown"},
};

using Pair = decltype(m)::value_type;

// Вставка может завершиться успехом…
auto nh1 = otherm.extract("sad");
assert(nh1.key() == "sad" && nh1.mapped() == "clown");
auto [it2, inserted2, nh2] = m.insert(std::move(nh1));
assert(*it2 == Pair("sad", "clown") && inserted2 == true && nh2.empty());

// ...или потерпеть неудачу из-за уже имеющихся элементов.
auto nh3 = otherm.extract("hello");
assert(nh3.key() == "hello" && nh3.mapped() == "dolly");
auto [it4, inserted4, nh4] = m.insert(std::move(nh3));
assert(*it4 == Pair("hello", "world") && inserted4 == false && !nh4.empty());

// Перезапись существующего элемента, это болезненная операция.
m.insert_or_assign(nh4.key(), nh4.mapped());

// Часто проще удалить элемент, который не удалось
// скопировать из-за существующего элемента.
m.erase(it4);
m.insert(std::move(nh4));
```

Метод `extract` возвращает объект, который иногда называют «дескриптором узла», по сути, указатель на внутреннюю структуру данных. Для доступа к составным частям элемента в `std::map` можно использовать методы `nh.key()` и `nh.mapped()` этого объекта (или `nh.value()` для доступа к единственной составной части в элементе `std::set`). Таким способом можно извлекать ключи, изменять их и вставлять обратно, не копируя и не перемещая фактических данных! В следующем примере демонстрируются «манипуляции», состоящие из вызовов метода `std::transform`:

```
std::map<std::string, std::string> m = {
  {"hello", "world"},
  {"quick", "brown"},
};
```

```
assert(m.begin()->first == "hello");
assert(std::next(m.begin())->first == "quick");

// Преобразовать в верхний регистр элемент {"quick", "brown"},
// без выделения памяти для промежуточных результатов.
auto nh = m.extract("quick");
std::transform(nh.key().begin(), nh.key().end(), nh.key().begin(),
::toupper);
m.insert(std::move(nh));

assert(m.begin()->first == "QUICK");
assert(std::next(m.begin())->first == "hello");
```

Как видите, интерфейс к этой функциональной возможности не так прост, как `lst.splice(it, otherlst)`; тонкости интерфейса – одна из причин, почему он был включен в стандартную библиотеку только в C++17. Хотелось бы обратить ваше внимание на следующее интересное обстоятельство: представьте, что вы извлекли узел из множества и затем, до того, как вам удалось вставить его в целевое множество, возбуждается исключение. Что случится с осиротевшим узлом – произойдет ли утечка? Как оказывается, проектировщики библиотеки предвидели эту возможность; если вызов деструктора дескриптора узла произойдет до того, как этот дескриптор будет вставлен на место, он благополучно освободит память, занимаемую узлом. То есть фактически `extract` (без последующего вызова `insert`) действует в точности как `erase`!

Хеши: std::unordered_set<T> и std::unordered_map<K, V>

Шаблонный класс `std::unordered_set` представляет хеш-таблицу с цепочками, то есть массив фиксированного размера, состоящий из элементов «корзин», каждый из которых хранит односвязный список элементов данных. По мере добавления новых элементов данных каждый из них помещается в односвязный список с «хешем», соответствующим значению элемента. Это почти то же самое, что `HashSet` в Java. В памяти этот контейнер хранится, как показано на рис. 4.10.

Хеш-таблицы детально описываются во множестве книг, и `std::unordered_set` даже с натяжкой нельзя назвать современным достижением; но, так как он устраняет некоторое количество указателей, этот контейнер обычно показывает более высокую производительность, чем древовидный `std::set`.

Чтобы избавиться от оставшихся указателей, можно заменить связные списки приемом с названием «открытая адресация». Его обсуждение далеко выходит за рамки этой книги, но вам определенно стоит познакомиться с ним, если производительность `std::unordered_set` покажется вам недостаточной.

```
std::unordered_set<char> m {42, 43, 44};
```

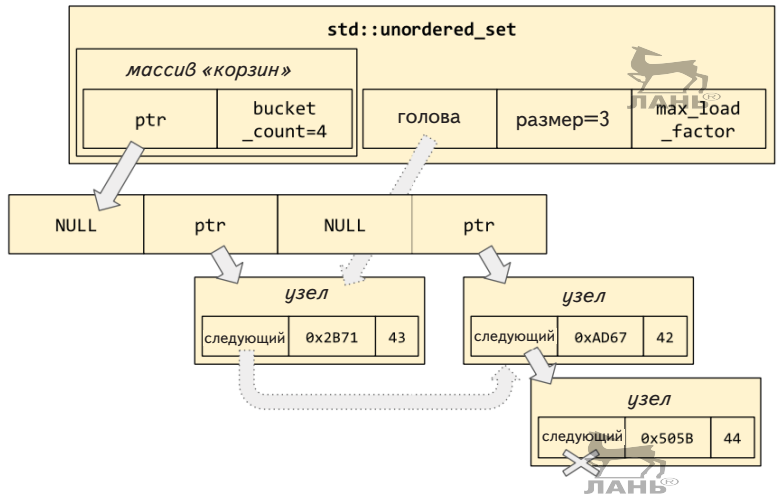


Рис. 4.10. Организация контейнера std::unordered_set в памяти

`std::unordered_set` задумывался как упрощенная замена `std::set`, поэтому поддерживает тот же интерфейс: `insert` и `erase`, плюс итерации с использованием `begin` и `end`. Но, в отличие от `std::set`, `std::unordered_set` хранит элементы не в порядке сортировки (это *неупорядоченная* коллекция) и поддерживает только прямые итераторы, тогда как `std::set` поддерживает двунаправленные. (Взгляните еще раз на рис. 4.10, где можно видеть указатели «следующий», но нет указателей «предыдущий», из-за чего итерации в обратном направлении в `std::unordered_set` невозможны.)

`std::unordered_map<K, V>` связан с `std::unordered_set<T>` так же, как `std::map<K, V>` связан с `std::set<T>`. То есть в памяти он хранится точно так же, только вместо ключей хранит пары ключ/значение.

Так же как `set` и `map`, которые принимают необязательный параметр с компаратором, `unordered_set` и `unordered_map` тоже принимают два дополнительных параметра: `Hash` (по умолчанию `std::hash<K>`) и `KeyEqual` (по умолчанию `std::equal_to<K>`, то есть `operator==`). Если передать другую хеш-функцию или другую функцию сравнения ключей, хеш-таблица будет использовать их вместо функций по умолчанию. Это может пригодиться для взаимодействий с традиционными классами на C++, не реализующими семантику значений или `operator==`:

```
class Widget {
public:
  virtual bool IsEqualTo(Widget const *b) const;
  virtual int GetHashValue() const;
};
```

```
struct myhash {
  size_t operator()(const Widget *w) const {
    return w->GetHashValue();
  }
};

struct myequal {
  bool operator()(const Widget *a, const Widget *b) const {
    return a->IsEqualTo(b);
  }
};

std::unordered_set<Widget *, myhash, myequal> s;
```

```
std::unordered_map<char, int> m {
    {42, 3},
    {43, 1},
    {44, 4},
};
```

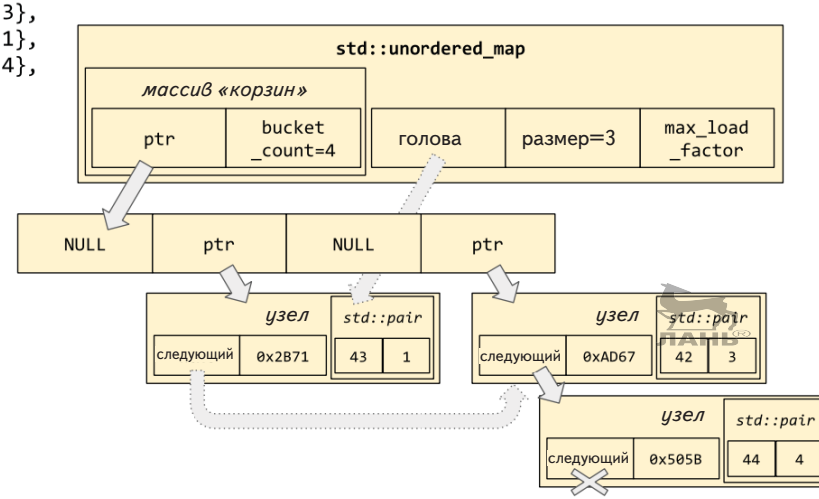


Рис. 4.11. Организация контейнера std::unordered_map в памяти

Фактор загрузки и списки в корзинах

Так же как HashSet в Java, `std::unordered_set` открывает доступ ко всей административной информации о корзинах. Но может так случиться, что вам никогда не понадобится обращаться к этим функциям!

- `s.bucket_count()` возвращает текущее количество корзин в массиве;
- `s.bucket(v)` возвращает индекс *i* корзины, где находится элемент `v`, если он имеется в данной коллекции `unordered_set`;
- `bucket_size(i)` возвращает количество элементов в *i*-й корзине; важно заметить, что всегда соблюдается условие `s.count(v) <= s.bucket_size(s.bucket(v))`;

- `s.load_factor()` возвращает `s.size()` / `s.bucket_count()` в виде числа типа `float`;
- `s.rehash(n)` увеличивает (или уменьшает) размер массива корзин до заданной величины `n`.

Возможно, вам покажется ненужной функция `load_factor`; что такого важного в отношении `s.size()` / `s.bucket_count()`, что ради этого была создана отдельная функция-член? На самом деле это механизм, который используется в `unordered_set` для масштабирования с ростом количества элементов. Каждый объект `s` типа `unordered_set` имеет значение `s.max_load_factor()`, указывающее, насколько большим может быть `s.load_factor()`. Если вставка нового элемента приведет к увеличению `s.load_factor()` выше `s.max_load_factor()`, тогда `s` перераспределит свой массив корзин и повторно выполнит хеширование элементов, чтобы уменьшить `s.load_factor()` ниже уровня `s.max_load_factor()`.

По умолчанию `s.max_load_factor()` имеет значение `1.0`. Вы можете задать другое значение `k` с вызовом перегруженной версии функции с одним параметром: `s.max_load_factor(k)`. Однако на практике этого почти никогда не требуется.

Одна из административных операций, которая действительно имеет смысл, – `s.reserve(k)`. Так же как `vec.reserve(k)` для векторов, эта функция-член `reserve` означает: «я планирую довести размер контейнера до размера `k`, поэтому выдели место для хранения будущих `k` элементов прямо сейчас». В случае с вектором это означает выделение памяти для массива с размером `k` элементов. В случае с `unordered_set` – выделение памяти для массива корзин с `k / max_load_factor()` указателями, чтобы после вставки `k` элементов (с ожидаемым числом коллизий) фактор загрузки не превышал `max_load_factor()`.

Откуда берется память?

На протяжении всей этой главы я на самом деле говорил вам не всю правду! Все контейнеры, представленные в этой главе, кроме `std::array`, принимают *еще один* необязательный параметр типа. Этот параметр называется диспетчером памяти – *распределителем* (allocator) – и определяет, откуда берется память для таких операций, как «перераспределение внутреннего массива» или «выделение памяти для нового узла в связном списке». `std::array` не нуждается в диспетчере памяти, потому что всю свою память содержит внутри себя; но все остальные типы контейнеров должны знать, где получить память.

По умолчанию в этом параметре передается стандартный тип `std::allocator<T>`, который подходит для большинства применений. Более подробно мы поговорим о диспетчерах памяти в главе 8 «Диспетчеры памяти».

Итоги

В этой главе мы узнали следующее.

- Контейнер управляет *владением* коллекции элементов.
- Все контейнеры STL являются шаблонными классами, которые параметризуются типом элементов и иногда другими соответствующими параметрами.
- Все контейнеры, кроме `std::array<T, N>`, можно параметризовать типом *диспетчера памяти*, чтобы указать механизм выделения и освобождения памяти.
- Контейнеры, использующие операцию сравнения, можно параметризовать типом *компаратора*. Подумайте об использовании прозрачных типов компараторов, таких как `std::less<>`, вместо гомогенных.

Пользуясь `std::vector`, помните о возможности перераспределения внутреннего массива и аннулировании указателей. В большинстве контейнеров наблюдается эффект аннулирования итераторов.

Философия стандартной библиотеки состоит в том, чтобы не поддерживать операции, неэффективные по своей природе (такие как `vector::push_front`); и поддерживать эффективные операции (такие как `list::splice`). Если придумали эффективную реализацию для конкретной операции, высока вероятность, что она уже включена в STL под каким-то именем; вам просто нужно выяснить – под каким.

Если есть сомнения, используйте `std::vector`. Используйте другие типы контейнеров, только когда вам действительно нужны определенные особенности, присущие им. В частности, старайтесь не пользоваться контейнерами на основе указателей (`set`, `map`, `list`), если вы не собираетесь использовать какие-то конкретные особенности (поддержание порядка сортировки; извлечение и объединение).

Лучшим источником дополнительной информации для вас станут онлайн-справочники, такие как cppreference.com.

Глава 5

Словарные типы

За последнее десятилетие неоднократно признавалось, что важнейшей ролью стандартного языка или стандартной библиотеки является определение *словарных типов*. Словарный тип – это тип, подразумевающий предоставление *единой основы*, обобщенного языка для выполнения операций в его предметной области.

Обратите внимание, что еще до появления C++ язык программирования C уже внес достойный вклад в словари для некоторых областей, предоставляя стандартные типы или синонимы для целочисленной (`int`) и вещественной (`double`) математики, для представления отметок времени в виде числа секунд от начала эпохи Unix (`time_t`) и счетчиков байтов (`size_t`).

В этой главе вы узнаете:

- историю развития словарных типов в C++, от `std::string` до `std::any`;
- как определяются *алгебраические типы данных*, *типы-произведения* и *типы-суммы*;
- как оперировать кортежами и перечислять варианты;
- роль `std::optional<T>` как «может быть T» или «еще не T»;
- `std::any` – алгебраический тип данных, как эквивалент «бесконечности»;
- как реализовать стирание типов и как использовать его в `std::any` и `std::function`, и какие врожденные ограничения оно имеет;
- некоторые ловушки `std::function` и познакомитесь со сторонними библиотеками, исправляющими их.

История std::string

Рассмотрим область строк символов; например, фразу `hello world`. В языке C строки представлялись типом `char *`:

```
char *greet(const char *name) {
  char buffer[100];
  snprintf(buffer, 100, "hello %s", name);
  return strdup(buffer);
```

```
}

void test() {
  const char *who = "world";
  char *hw = greet(who);
  assert(strcmp(hw, "hello world") == 0);
  free(hw);
}
```

Этот подход благополучно использовался некоторое время, но работа с простым типом `char *` было сопряжена с некоторыми проблемами для пользователей языка и создателей сторонних библиотек и процедур. Во-первых, язык C настолько стар, что в нем с самого начала отсутствовал спецификатор `const`, а значит, некоторые старые подпрограммы могут ожидать получить строки в виде `char *`, а более новые – в виде `const char *`. Во-вторых, тип `char *` не несет в себе длину строки, поэтому некоторые функции требуют передачи указателя и длины, а некоторые – только указателя и просто не предполагают встретить внутри строки байты со значением `'\0'`.

Самым большим недостатком типа `char *` является отсутствие *механизмов управления жизненным циклом и владением* (как обсуждалось в начале главы 4 «Зоопарк контейнеров»). Когда функция на C требует от вызывающего кода передать ей строку, она принимает `char *` и обычно оставляет за вызывающим кодом право управлять владением строкой символов. А что, если функция должна *вернуть* строку? Тогда она должна вернуть `char *` и надеяться, что вызывающий код не забудет освободить ее (`strdup`, `asprintf`), или принять от вызывающего кода буфер и надеяться, что он достаточно большой, чтобы в нем уместился результат (`sprintf`, `snprintf`, `strcat`). Сложность управления владением строк в C (и в первых версиях C++) была настолько высока, что для решения этой проблемы появилось множество «строковых библиотек»: `QString` в Qt, `GString` в glib и т. д.

В этот хаос в 1998 году C++ принес чудо: *стандартный* класс строк! Новый тип `std::string` естественным образом инкапсулировал в себе байты строки *и* ее длину. Получил возможность правильно обрабатывать нулевые байты. Организовал поддержку прежде сложных операций, таких как `hello + world`, внутренне выделяя столько памяти, сколько необходимо. И благодаря идиоме RAII он не страдал проблемой утечки памяти и не вызывал путаницу в отношении владения внутренними байтами. Но самое замечательное – он допускал неявное преобразование из типа `char *`:

```
std::string greet(const std::string& name) {
  return "hello " + name;
}

void test() {
  std::string who = "world";
  assert(greet(who) == "hello world");
}
```

Теперь функции на C++, обрабатывающие строки (как `greet()` в предыдущем примере), могут принимать параметры и возвращать результаты типа `std::string`. Более того, благодаря *стандартизации* строкового типа можно быть уверенными на несколько лет вперед в правильности выбора сторонней библиотеки для интеграции в свои проекты, если любая из ее функций, принимающая строки (имена файлов, сообщения об ошибках и вообще все, что угодно), принимает тип `std::string`. Появление новой единой основы в лице `std::string` дало возможность взаимодействовать более действенно и эффективно.

Маркировка ссылочных типов с reference_wrapper

В C++03 появился еще один библиотечный тип – `std::reference_wrapper<T>`. Он имел очень простую реализацию:

```
namespace std {
  template<typename T>
  class reference_wrapper {
    T *m_ptr;
  public:
    reference_wrapper(T& t) noexcept : m_ptr(&t) {}

    operator T& () const noexcept { return *m_ptr; }
    T& get() const noexcept { return *m_ptr; }
  };

  template<typename T>
  reference_wrapper<T> ref(T& t);
} // namespace std
```

`std::reference_wrapper` имеет немного отличное назначение, чем другие словарные типы, такие как `std::string` и `int`; его назначение – служить «маркером» для значений, которые должны действовать как ссылки в контекстах, где передача обычных ссылок C++ работает не так, как нам хотелось бы:

```
int result = 0;
auto task = [](int& r) {
  r = 42;
};

// Попытка использовать обычную ссылку не компилируется.
//std::thread t(task, result);

// Правильная передача result "по ссылке" в новый поток.
std::thread t(task, std::ref(result));
```

Конструктор `std::thread` предусматривает особые случаи для обработки параметров `reference_wrapper` преобразованием их в обычные ссылки. Те

же специальные случаи применяются к функциям в стандартной библиотеке `make_pair`, `make_tuple`, `bind`, `invoke` и ко всем, основанным на `invoke` (таким как `std::apply`, `std::function::operator()` и `std::async`).

C++11 и алгебраические типы

По мере формирования C++11 все более обоснованным выглядело мнение, что пришла пора определить словарные *алгебраические типы данных*. Алгебраические типы занимают естественное положение в парадигме функционального программирования. Ключевая идея состоит в том, чтобы представить предметную область типа – то есть множество возможных значений. Для простоты вообразите типы перечислений в C++, потому что они позволяют легко рассуждать о количестве разных значений, которые объект типа перечисления может принимать:

```
enum class Color {
  RED = 1,
  BLACK = 2,
};

enum class Size {
  SMALL = 1,
  MEDIUM = 2,
  LARGE = 3,
};
```


Можно ли на основе типов `Color` и `Size` создать новый тип данных, экземпляры которого смогут иметь любое из 2 × 3 = 6 значений? Да; этот тип будет служить представлением «по одному из каждого» `Color` и `Size`, и такие типы часто называют *типами-произведениями*, потому что множеством возможных значений является *декартово произведение* возможных значений его элементов.

А можно ли определить тип, экземпляры которого смогут иметь любое из 2 + 3 = 5 разных значений? Тоже да; этот тип будет служить представлением «либо `Color`, либо `Size`, но никогда вместе» и называется *типами-суммами*. (Как ни странно, но математики не используют понятие *декартовой суммы* для обозначения этой идеи.)

В функциональных языках, таких как Haskell, эти два типа можно определить так:

```
data SixType = ColorandSizeOf Color Size;
data FiveType = ColorOf Color | SizeOf Size;
```

В C++ они определяются так:

```
using sixtype = std::pair<Color, Size>;
using fivetype = std::variant<Color, Size>;
```

Шаблонный класс `std::pair<A, B>` представляет упорядоченную пару элементов: первым следует элемент типа `A`, а за ним – элемент типа `B`. Очень похоже на простую структуру с двумя элементами, только в данном случае не требуется писать определение структуры:

```
template<class A, class B>
struct pair {
  A first;
  B second;
};
```

Обратите внимание, что различия между `std::pair<A, A>` и `std::array<A, 2>` носят чисто косметический характер. Можно сказать, что `pair` – это гетерогенная версия `array` (если не учитывать, что `pair` может хранить только два элемента).

Работа с std::tuple

В стандарте C++11 появился полноценный гетерогенный массив: кортеж `std::tuple<Ts...>`. Кортеж с двумя параметрами типов – например, `tuple<int, double>` – ничем не отличается от `pair<int, double>`. Но кортежи могут хранить намного больше элементов, не только пару; несмотря на то что волшебство вариативных шаблонов (шаблонов с переменным числом параметров) в C++11 позволяет определять кортежи с тремя, четырьмя, пятью и т. д. элементами, для типа выбрано общее универсальное имя `tuple`. Например, `tuple<int, int, char, std::string>` является аналогом структуры с членами `int`, `int`, `char` и `std::string`.

Поскольку элементы кортежа имеют разные типы, нельзя использовать «обычный» `operator[](size_t)` для доступа к элементам по индексам, которые могут меняться во время выполнения. Вместо этого мы должны сообщить компилятору *во время компиляции*, к какому элементу кортежа хотим обратиться, чтобы компилятор смог определить тип выражения. Для принудительной передачи информации системе типов на этапе компиляции в C++ выбран способ ввода через шаблонные параметры. То есть чтобы получить доступ к первому элементу кортежа `t`, мы должны вызвать `std::get<0>(t)`. Чтобы получить доступ ко второму элементу, нужно вызвать `std::get<1>(t)` и т. д.

Это типичный способ работы с `std::tuple` – там, где контейнеры с элементами одного типа предоставляют *функции-члены* для доступа к ним, гетерогенные алгебраические типы предоставляют *шаблонные функции*.

Но, вообще говоря, вам не часто придется манипулировать кортежами. Их главное назначение не попадает в сферу метапрограммирования шаблонов, они дают лишь упрощенный способ связать воедино несколько значений в контексте, где требуется единственное значение. Например, вспомните `std::tie` из примера в разделе «Простейший контейнер» в главе 4 «Зоопарк контейнеров». Это недорогой способ объединения произвольного количества значений

в единое целое, которое может сравниваться лексикографически с помощью operator<. Результат лексикографического сравнения при этом зависит от порядка, в каком происходит объединение значений:

```
using Author = std::pair<std::string, std::string>;
std::vector<Author> authors = {
  {"Fyodor", "Dostoevsky"},
  {"Sylvia", "Plath"},
  {"Vladimir", "Nabokov"},
  {"Douglas", "Hofstadter"},
};

// Сортировать по имени, потом по фамилии.
std::sort(
  authors.begin(), authors.end(),
  [](auto&& a, auto&& b) {
    return std::tie(a.first, a.second) < std::tie(b.first, b.second);
  }
);
assert(authors[0] == Author("Douglas", "Hofstadter"));

// Сортировать по фамилии, потом по имени.
std::sort(
  authors.begin(), authors.end(),
  [](auto&& a, auto&& b) {
    return std::tie(a.second, a.first) < std::tie(b.second, b.first);
  }
);
assert(authors[0] == Author("Fyodor", "Dostoevsky"));
```

Причина дешевизны std::tie в том, что фактически она не копирует свои аргументы, а создает кортеж *ссылок* на них. Это приводит ко второму типичному случаю использования std::tie: моделированию «множественного присваивания», имеющегося в таких языках, как Python:

```
std::string s;
int i;

// Присвоить значения двум переменным, s и i, одновременно.
std::tie(s, i) = std::make_tuple("hello", 42);
```

Обратите внимание, что слово «одновременно» в предыдущем комментарии не имеет никакого отношения ни к конкурентному выполнению (см. главу 7 «Конкуренция»), ни к очередности любых возникающих побочных эффектов; я просто имел в виду, что оба значения присваиваются в одной инструкции присваивания.

Как иллюстрирует предыдущий пример, `std::make_tuple(a, b, c...)` можно использовать для создания кортежа значений; то есть `make_tuple` конструирует копии значений своих аргументов, а не просто сохраняет их адреса.

Наконец, в C++17 мы получили возможность использовать автоматическое определение параметра шаблонного конструктора, чтобы можно было просто записать `std::tuple(a, b, c...)`; но, вероятно, такой возможности лучше избегать, не зная точно особенностей ее поведения. Единственное, что делает автоматический вывод параметра шаблона иначе, чем `std::make_tuple`, – он сохраняет аргументы `std::reference_wrapper`, не преобразуя их в обычные ссылки C++:

```
auto [i, j, k] = std::tuple{1, 2, 3};

// make_tuple преобразует reference_wrapper...
auto t1 = std::make_tuple(i, std::ref(j), k);
static_assert(std::is_same_v< decltype(t1),
  std::tuple<int, int&, int>
>);

// ...тогда как автоматический вывод конструктора нет.
auto t2 = std::tuple(i, std::ref(j), k);
static_assert(std::is_same_v< decltype(t2),
  std::tuple<int, std::reference_wrapper<int>, int>
>);
```

Манипулирование значениями кортежа

Большинство из этих функций и шаблонов удобно использовать только в контексте метапрограммирования шаблонов; вам едва ли придется использовать их в повседневной работе:

- `std::get<I>(t)`: возвращает ссылку на `I`-й элемент в `t`;
- `std::tuple_size_v<decltype(t)>`: возвращает *размер* указанного кортежа. Так как это константное свойство типа кортежа времени компиляции, оно выражается как шаблон переменной, параметризованный этим типом. Если вы предпочитаете использовать синтаксис, который смотрится более естественно, можете написать вспомогательную функцию любым из следующих способов:

```
template<class T>
constexpr size_t tuple_size(T&&)
{
  return std::tuple_size_v<std::remove_reference_t<T>>;
}

template<class... Ts>
constexpr size_t simpler_tuple_size(const std::tuple<Ts...>&)
```

```
{
  return sizeof...(Ts);
}
```

- `std::tuple_element_t<I, decltype(t)>`: возвращает *тип* `I`-го элемента данного типа кортежа. И снова стандартная библиотека дает более неудобный способ получения этой информации, чем основной язык. Вообще, чтобы определить тип `I`-го элемента кортежа, достаточно воспользоваться выражением `decltype(std::get<I>(t))`;
- `std::tuple_cat(t1, t2, t3...)`: объединяет указанные кортежи, выстраивая их друг за другом;
- `std::forward_as_tuple(a, b, c...)`: создает кортеж ссылок, в точности как `std::tie`; но если `std::tie` требует ссылки *lvalue*, то `std::forward_as_tuple` принимает на входе любые ссылки и переправляет их в кортеж так, что потом их можно извлечь с помощью `std::get<I>(t)...`:

```
template<typename F>
void run_zeroarg(const F& f);

template<typename F, typename... Args>
void run_multiarg(const F& f, Args&&... args)
{
  auto fwd_args =
    std::forward_as_tuple(std::forward<Args>(args)...);
  auto lambda = [&f, fwd_args]() {
    std::apply(f, fwd_args);
  };
  run_zeroarg(f);
}
```

Замечание об именованных классах

Как мы видели в главе 4 «Зоопарк контейнеров», когда сравнивали `std::array<double, 3>` со структурой `struct Vec3`, использование шаблонного класса из STL может сократить затраты времени на разработку и устранить источники ошибок за счет повторного использования проверенных компонентов STL; но может сделать код трудночитаемым или дать вашему типу излишнюю функциональность. В нашем примере в главе 4 «Зоопарк контейнеров» выбор `std::array<double, 3>` оказался неудачным по сравнению с `Vec3`, потому что реализует нежелательный `operator<`.

Непосредственное использование любых алгебраических типов (`tuple`, `pair`, `optional` или `variant`) в ваших интерфейсах и API, скорее всего, является ошибкой. Вы заметите, что ваш код выглядит проще и понятнее, если писать именованные классы для своих предметных типов, даже если – и *особенно если* – они являются лишь тонкими обертками вокруг алгебраических типов.

Выражение альтернатив с помощью std::variant

Если `std::tuple<A,B,C>` является *типом-произведением*, то `std::variant <A,B,C>` – это *тип-сумма*. Вариант позволяет хранить или `A`, или `B`, или `C`, но не больше (и не меньше). Иначе это понятие называется *размеченное объединение* (discriminated union), потому что вариант во многом напоминает обычное объединение `union` в C++; но, в отличие от него, вариант всегда может сообщить, какой из его элементов – `A`, `B` или `C` – «активен» в данный момент. Официально эти элементы называются «альтернативами», потому что активным может быть только один из них:

```
std::variant<int, double> v1;

v1 = 1; // активировать член "int"
assert(v1.index() == 0);
assert(std::get<0>(v1) == 1);

v1 = 3.14; // активировать член "double"
assert(v1.index() == 1);
assert(std::get<1>(v1) == 3.14);
assert(std::get<double>(v1) == 3.14);

assert(std::holds_alternative<int>(v1) == false);
assert(std::holds_alternative<double>(v1) == true);

assert(std::get_if<int>(&v1) == nullptr);
assert(*std::get_if<double>(&v1) == 3.14);
```

Так же как в случае с `tuple`, получить конкретный элемент из варианта `variant` можно с помощью `std::get<I>(v)`. Если все альтернативы объекта `variant` различны (что является наиболее типичным случаем использования, если не увлекаться глубоким метапрограммированием), тогда можно использовать версию `std::get<T>(v)` с типами вместо индексов – например, в предыдущем фрагменте кода вместо `std::get<0>(v1)` можно было бы использовать вызов `std::get<int>(v1)`, потому что нулевая альтернатива в варианте `v1` имеет тип `int`. В отличие от `tuple`, однако, вызов `std::get` с вариантом может потерпеть неудачу! Например, вызов `std::get<double>(v1)` для `v1`, хранящего в данный момент значение типа `int`, возбудит исключение `std::bad_variant_access`. `std::get` имеет версию `std::get_if`, которая не возбуждает исключений. Как показано в предыдущем примере, `get_if` возвращает указатель на заданную альтернативу, если она активна, и `null` в ином случае. То есть следующий фрагмент кода является полным эквивалентом предыдущего:

```
// Худший...
try {
```

```
  std::cout << std::get<int>(v1) << std::endl;
} catch (const std::bad_variant_access&) {}

// Все еще плохой...
if (v1.index() == 0) {
  std::cout << std::get<int>(v1) << std::endl;
}

// Немного лучше...
if (std::holds_alternative<int>(v1)) {
  std::cout << std::get<int>(v1) << std::endl;
}

// ...Лучший.
if (int *p = std::get_if<int>(&v1)) {
  std::cout << *p << std::endl;
}
```

Чтение вариантов

Предыдущий пример демонстрирует, что для переменной `std::variant <int, double> v` вызов `std::get<double>(v)` возвращает текущее значение, если в этот момент она хранит значение типа `double` и генерирует исключение, если она хранит значение типа `int`. Это может показаться странным: тип `int` легко преобразуется в тип `double`, тогда почему функция не может просто вернуть преобразованное значение?

Мы легко можем получить такое поведение, но не от `std::get`. Для этого выразим свое желание немного иначе: «У меня есть переменная типа `variant`. Если сейчас она хранит значение типа `double`, назвать его `d` и дать возможность получить его как `double(d)`. Если она хранит значение типа `int`, назвать его `i` и дать возможность получить его как `double(i)`». То есть у нас имеется перечень желаемых реакций и нам хотелось бы, чтобы вызывалась та реакция, которая соответствует типу альтернативы, хранящейся в `v`. Стандартная библиотека реализует этот алгоритм под немного странным именем `std::visit`:

```
struct Visitor {
  double operator()(double d) { return d; }
  double operator()(int i) { return double(i); }
  double operator()(const std::string&) { return -1; }
};

using Var = std::variant<int, double, std::string>;

void show(Var v)
{
  std::cout << std::visit(Visitor{}, v) << std::endl;
}

void test()
```

```
{
  show(3.14);
  show(1);
  show("hello world");
}
```

В общем случае все виды чтения варианта с помощью `visit`, которые мы определили, имеют фундаментальное сходство. Так как язык C++ поддерживает перегрузку функций и операторов, мы обычно можем выразить похожие виды поведения, используя идентичный синтаксис. А если мы можем выразить их с применением идентичного синтаксиса, значит, можем завернуть их в шаблонную функцию или – наиболее типичный подход – обобщенное лямбда выражение C++14, например:

```
std::visit([](const auto& alt) {
  if constexpr (std::is_same_v<decltype(alt), const std::string&>) {
    std::cout << double(-1) << std::endl;
  } else {
    std::cout << double(alt) << std::endl;
  }
}, v);
```

Обратите внимание, что здесь используется конструкция `if constexpr`, появившаяся в C++17, которая обрабатывает один из случаев, принципиально отличающийся от других. Выбор решения, использовать ли явное включение `decltype`, как здесь, или определить вспомогательный класс, как `Visitor` в предыдущем примере, и довериться механизму разрешения в выборе верной перегруженной версии `operator()`, во многом является делом личного вкуса.

Имеется также вариативная версия `std::visit`, принимающая два, три и даже большее количество объектов `variant` одного или разных типов. Эту версию `std::visit` можно использовать для реализации своеобразной «многоцелевой диспетчеризации», как показано в следующем примере. Однако эта версия `std::visit` едва ли понадобится вам, если вы не занимаетесь метапрограммированием всерьез:

```
struct MultiVisitor {
  template<class T, class U, class V>
  void operator()(T, U, V) const { puts("wrong"); }

  void operator()(char, int, double) const { puts("right!"); }
};

void test()
{
```

```
    std::variant<int, double, char> v1 = 'x';
    std::variant<char, int, double> v2 = 1;
    std::variant<double, char, int> v3 = 3.14;
    std::visit(MultiVisitor{}, v1, v2, v3); // выведет "right!"
}
```

О make_variant и семантике типа-значения

Как мы знаем, объект `tuple` можно создать с помощью `std::make_tuple`, пару – с помощью `make_pair`. Поэтому уместным выглядит вопрос: «Существует ли `make_variant`?» Как оказывается, такой функции нет. Главная причина в том, что, в отличие от типов-произведений `tuple` и `pair`, тип `variant` является типом-суммой. Создавая кортеж, мы всегда указываем все *n* значений для его элементов, благодаря чему типы элементов легко выводятся. Однако, создавая вариант, мы можем указать только одно из возможных значений, например типа `A`, поэтому компилятор не сможет создать объект `variant<A,B,C>`, не имея возможности идентифицировать типы `B` и `C`. То есть функция `my::make_variant<A,B,C>(a)` теряет всякий смысл, учитывая, что фактический конструктор класса записывается короче: `std::variant<A,B,C>(a)`.

Мы уже упоминали вторую причину существования `make_pair` и `make_tuple`: они автоматически преобразуют специальный словарный тип `std::reference_wrapper<T>` в `T&`, благодаря чему `std::make_pair(std::ref(a), std::cref(b))` создаст объект типа `std::pair<A&, const B&>`. Объекты «пара ссылок» или «кортеж ссылок» проявляют очень странное поведение: их можно сравнивать и копировать, используя обычную семантику, но когда вместо «привязки» элементов (чтобы они ссылались на объекты справа) выполняется присваивание объекту того же типа, оператор присваивания фактически изменяет значения объектов, на которые указывают ссылки. Как было показано в разделе «Работа с std::tuple», эта преднамеренная странность позволяет нам использовать `std::tie` как своеобразную инструкцию «множественного присваивания».

Поэтому еще одна причина, по которой можно ожидать или желать увидеть функцию `make_variant` в стандартной библиотеке, заключается в возможности преобразования ссылок. Однако это тоже довольно спорный вопрос по одной простой причине – стандарт запрещает создавать варианты из элементов ссылочных типов! Далее в этой главе мы увидим, что типам `std::optional` и `std::any` также запрещено хранить ссылочные типы. (Однако `std::variant<std::reference_wrapper<T>, ...>` является вполне допустимым.) Такой запрет обусловлен тем, что создатели библиотеки не пришли к единому мнению о том, что должен означать вариант ссылок. И, если уж на то пошло, что должно означать понятие *кортеж ссылок*! Единственная причина, почему сегодня в языке присутствует поддержка кортежа ссылок, – `std::tie` выглядела отличной идеей в 2011 году. В 2017 году не нашлось желающих усугубить путаницу введением вариантов, необязательных и «произвольных» ссылок.

Итак, мы установили, что `std::variant<A,B,C>` всегда хранит точно одно значение типа `A`, `B` или `C` – не больше и не меньше. Однако технически это

утверждение не совсем верно. *В некоторых очень необычных обстоятельствах* можно сконструировать вариант, вообще не имеющий значения. Единственный путь добиться этого – сконструировать вариант со значением типа `A`, а затем присвоить ему значение типа `B`, в результате чего значение `A` будет благополучно уничтожено, а конструктор `B` возбудит исключение и значение `B` никогда не будет помещено в вариант. Когда такое происходит, объект варианта оказывается в состоянии, известном как «без значения из-за исключения» (valueless by exception):


```
struct A {
  A() { throw "ha ha!"; }
};
struct B {
  operator int () { throw "ha ha!"; }
};
struct C {
  C() = default;
  C& operator=(C&&) = default;
  C(C&&) { throw "ha ha!"; }
};

void test()
{
  std::variant<int, A, C> v1 = 42;

  try {
    v1.emplace<A>();
  } catch (const char *haha) {}
  assert(v1.valueless_by_exception());

  try {
    v1.emplace<int>(B());
  } catch (const char *haha) {}
  assert(v1.valueless_by_exception());
}
```

Этого никогда не произойдет с вами, если вы не пишете код, в котором ваши конструкторы или операторы преобразования возбуждают исключения. Кроме того, если вместо `emplace` использовать `operator=`, можно избежать вариантов без значений во всех случаях, кроме ситуаций, когда имеется конструктор перемещения, способный возбудить исключение:

```
v1 = 42;

// Конструктор справа от оператора присваивания возбудит исключение;
// но это не затронет вариант.
try { v1 = A(); } catch (...) {}
assert(std::get<int>(v1) == 42);

// В этом случае тоже.
```

```
try { v1 = B(); } catch (...) {}
assert(std::get<int>(v1) == 42);

// Но исключение в конструкторе перемещения может все испортить.
try { v1 = C(); } catch (...) {}
assert(v1.valueless_by_exception());
```

Как рассказывалось при обсуждении `std::vector` в главе 4 «Зоопарк контейнеров», конструкторы перемещения ваших типов всегда должны снабжаться спецификатором `noexcept`; то есть если вы неуклонно будете придерживаться этого правила, вы можете вообще никогда не столкнуться с необходимостью использовать `valueless_by_exception`.

В любом случае, когда вариант находится в этом состоянии, его метод `index()` вернет `size_t(-1)` (константу, известную также как `std::variant_npos`), и любая попытка вызвать `std::visit` возбудит исключение `std::bad_variant_access`.

Задержка инициализации с помощью std::optional

Возможно, вы уже подумали, что одно из возможных применений `std::variant` – представление понятия «может быть, у меня есть объект, а может быть, нет». Например, состояние «возможно, нет» можно представить с использованием стандартного тега `std::monostate`:

```
std::map<std::string, int> g_limits = {
  { "memory", 655360 }
};

std::variant<std::monostate, int>
get_resource_limit(const std::string& key)
{
  if (auto it = g_limits.find(key); it != g_limits.end()) {
    return it->second;
  }
  return std::monostate{};
}

void test()
{
  auto limit = get_resource_limit("memory");
  if (std::holds_alternative<int>(limit)) {
    use( std::get<int>(limit) );
  } else {
    use( some_default );
  }
}
```

Спешу вас обрадовать, что это не лучший способ достичь поставленной цели! Стандартная библиотека поддерживает словарный тип `std::optional<T>` специально для понятия «может быть, есть, а может быть, нет».

```
std::optional<int>
get_resource_limit(const std::string& key)
{
  if (auto it = g_limits.find(key); it != g_limits.end()) {
    return it->second;
  }
  return std::nullopt;
}

void test()
{
  auto limit = get_resource_limit("memory");
  if (limit.has_value()) {
    use( *limit );
  } else {
    use( some_default );
  }
}
```

В логике алгебраических типов `std::optional<T>` – это тип-сумма: он имеет ровно столько же значений, сколько имеет тип `T`, плюс один. Это дополнительное значение называется «null», «пустое» или «незанятое» состояние и представляется в исходном коде специальной константой `std::nullopt`.

Не путайте константу `std::nullopt` с похожей на нее `std::nullptr`! Они не имеют ничего общего, кроме того что обе представляют «ничто».

В отличие от `std::tuple` и `std::variant` с их мешаниной свободных функций (не являющихся членами), класс `std::optional<T>` имеет множество удобных функций-членов. `o.has_value()` вернет `true`, если необязательный объект `o` в настоящий момент хранит значение типа `T`. Состояние «имеет значение» часто называют «занятым» состоянием; необязательный объект, содержащий значение, «занят», а необязательный объект в пустом состоянии «не занят».

Операторы сравнения `==`, `!=`, `<`, `<=`, `>` и `>=` перегружены для `optional<T>`, если имеют смысл для `T`. При этом, сравнивая два экземпляра `optional` или `optional` со значением типа `T`, помните, что экземпляр `optional` в незанятом состоянии всегда считается «меньше», чем любое действительное значение `T`.

`bool(o)` является синонимом `o.has_value()`, а `!o` – синонимом `!o.has_value()`. Лично я рекомендую всегда использовать `has_value`, потому что это не влечет дополнительных накладных расходов во время выполнения; единст-

венное отличие – удобочитаемость кода. Если вы решите использовать сокращенную форму преобразования в логическое значение, помните, что для `std::optional<bool>` операции `o == false` и `!o` означают совершенно разные вещи!

`o.value()` возвращает ссылку на значение, содержащееся в `o`. Если в данный момент объект `o` не занят, тогда `o.value()` возбудит исключение `std::bad_optional_access`.

`*o` (используется перегруженный унарный `operator*`) возвращает ссылку на значение в `o` без проверки на занятость. Если в данный момент экземпляр `o` не занят, поведение `*o` не определено, так же как поведение `*p` для пустого указателя. Эту особенность легко запомнить, если обратить внимание, что стандартная библиотека C++ применяет символы пунктуации для реализации наиболее эффективных операций, содержащих меньше проверок. Например, `std::vector::operator[]` выполняет меньше проверок на выход за границы, чем `std::vector::at()`. Следуя той же логике, `std::optional::operator*` выполняет меньше проверок, чем `std::optional::value()`.

`o.value_or(x)` возвращает копию значения, содержащегося в `o`, или, если объект `o` не занят, копию значения `x`, преобразованного в тип `T`. С помощью `value_or` можно заменить предыдущий пример единственной строкой, сохранив простоту и читаемость кода:

```
std::optional<int> get_resource_limit(const std::string&);

void test() {
  auto limit = get_resource_limit("memory");
  use( limit.value_or(some_default) );
}
```

В предыдущих примерах демонстрируется использование `std::optional<T>` для обработки случая «возможно, T» (как тип возвращаемого значения функции или как тип параметра). Другой часто встречающийся случай использования `std::optional<T>` – обработка ситуации «возможно, не T», как член данных класса. Например, допустим, что у нас есть некоторый тип `L`, не имеющий конструктора по умолчанию, такой как тип замыкания, производимый лямбда-выражением:

```
auto make_lambda(int arg) {
  return [arg](int x) { return x + arg; };
}
using L = decltype(make_lambda(0));

static_assert(!std::is_default_constructible_v<L>);
static_assert(!std::is_move_assignable_v<L>);
```

Тогда класс с членом этого типа также будет считаться не имеющим конструктора по умолчанию:

```
class ProblematicAdder {
  L fn_;
};

static_assert(!std::is_default_constructible_v<ProblematicAdder>);
```

Но, если обозначить тип нашего члена класса как `std::optional<L>`, мы сможем использовать класс в контекстах, где требуется конструктор по умолчанию:

```
class Adder {
  std::optional<L> fn_;
public:
  void setup(int first_arg) {
    fn_.emplace(make_lambda(first_arg));
  }
  int call(int second_arg) {
    // возбудит исключение, если прежде не вызвать setup()
    return fn_.value()(second_arg);
  }
};

static_assert(std::is_default_constructible_v<Adder>);

void test() {
  Adder adder;
  adder.setup(4);
  int result = adder.call(5);
  assert(result == 9);
}
```

Обратите внимание, что если по какой-то причине понадобится получить неопределенное поведение `call()` вместо возбуждения исключения, можно просто заменить `fn_.value()` на `*fn_`.

Реализовать такое поведение без `std::optional` было бы очень сложно. Для этого можно применить размещающую (placement) форму оператора `new` или использовать объединение, но при этом вам фактически придется самостоятельно реализовать половину класса `optional`. Использование `std::optional` на этом фоне выглядит намного лучше!

`std::optional` – действительно одно из самых больших достижений C++17, и знакомство с этим типом принесет вам огромную пользу.

Теперь перейдем от типа `optional`, который можно назвать вариантом `variant`, ограниченным единственным типом, к другой крайности: алгебраическому типу данных, эквивалентному бесконечности.

И снова variant

Тип данных `variant` хорош для представления простых альтернатив, но, например, для представления *рекурсивных* типов данных, появившихся в C++17, таких как списки JSON, он не подходит. То есть следующий код на C++17 не будет компилироваться:

```
using JSONValue = std::variant<
  std::nullptr_t,
  bool,
  double,
  std::string,
  std::vector<JSONValue>,
  std::map<std::string, JSONValue>
>;
```

Есть несколько обходных решений этой проблемы. Наиболее надежный и правильный – продолжать использовать тип `boost::variant` из библиотеки C++11 Boost, который поддерживает, в частности, рекурсивные вариантные типы через тип-маркер `boost::recursive_variant_`:

```
using JSONValue = boost::variant<
  std::nullptr_t,
  bool,
  double,
  std::string,
  std::vector<boost::recursive_variant_>,
  std::map<std::string, boost::recursive_variant_>
>;
```

Также проблему можно решить, определив новый класс `JSONValue`, который будет **включать** (HAS-A) или **наследовать** (IS-A) `std::variant` рекурсивного типа.

В следующем примере я выбрал решение на основе включения (HAS-A), а не наследования (IS-A); наследование неполиморфных типов из стандартной библиотеки – практически всегда плохая идея.

Поскольку в C++ допускаются опережающие ссылки на классы, следующий код скомпилируется:

```
struct JSONValue {
  std::variant<
    std::nullptr_t,
    bool,
```

```
    double,
    std::string,
    std::vector<JSONValue>,
    std::map<std::string, JSONValue>
  > value_;
};
```


Далее рассмотрим другой алгебраический тип из стандартной библиотеки, еще более мощный, чем `variant`.

Бесконечное число альтернатив с std::any

Перефразируем известное выражение Генри Форда: объект типа `std::variant<A, B, C>` может хранить значение любого типа, при условии что это тип `A`, `B` или `C`. А теперь представьте, что нам понадобилось обеспечить хранение *по-настоящему* любого типа. Такое может понадобиться, например, когда программа загружает дополнительные плагины, содержащие новые типы, неизвестные заранее, – мы не сможем указать эти типы в своем варианте `variant` – или в ситуации с «рекурсивным типом данных», описанной в предыдущем разделе.

Для этих случаев в стандартной библиотеке C++17 определен алгебраический тип `std::any` – разновидность «бесконечности». Это контейнер (см. главу 4 «Зоопарк контейнеров») для единственного объекта абсолютно любого типа. Контейнер может быть пустым или содержать объект. Объект `any` поддерживает следующие базовые операции:

- получить признак наличия объекта в контейнере;
- поместить новый объект в контейнер (уничтожив прежний, если имеется);
- получить тип хранящегося объекта;
- получить хранящийся объект, правильно указав его тип.

Вот как выглядят в коде первые три операции:

```
std::any a; // создать пустой контейнер

assert(!a.has_value());

a = std::string("hello");
assert(a.has_value());
assert(a.type() == typeid(std::string));

a = 42;
assert(a.has_value());
assert(a.type() == typeid(int));
```

Четвертая операция немного сложнее. Она записывается как `std::any_cast`, напоминает `std::get` для вариантов и имеет две разновидности: разновид-

ность в стиле `std::get`, которая возбуждает исключение `std::bad_any_cast` в случае неудачи, и разновидность в стиле `std::get_if`, которая возвращает пустой указатель:

```
if (std::string *p = std::any_cast<std::string>(&a)) {
  use(*p);
} else {
  // обработать ошибку
}
try {
std::string& s = std::any_cast<std::string&>(a);
use(s);
} catch (const std::bad_any_cast&) {
  // обработать ошибку
}
```

Обратите внимание, что в любом случае требуется указать тип, который требуется извлечь из объекта `any`. Указав неверный тип, вы получите исключение или пустой указатель. Нет никакой возможности выразить желание: «Дай мне хранимый объект независимо от его типа», – потому что тип такого выражения неизвестен.

Столкнувшись с подобной проблемой с `std::variant` в предыдущем разделе, мы решили использовать `std::visit`, чтобы получить некоторый обобщенный код выбора хранящейся альтернативы. К сожалению, для типа `any` нет алгоритма, эквивалентного `std::visit`. Причина проста и непреодолима: раздельная компиляция. Представьте, что в одном файле *a.cc* я написал:

```
template<class T> struct Widget {};

std::any get_widget() {
  return std::make_any<Widget<int>>();
}
```

А в другом файле *b.cc* (который может компилироваться в другой плагин, библиотеку *.dll* или *.so*):

```
template<class T> struct Widget {};

template<class T> int size(Widget<T>& w) {
  return sizeof w;
}

void test()
{
  std::any a = get_widget();
  int sz = hypothetical_any_visit([](auto&& w){
    return size(w);
  }, a);
```

```
  assert(sz == sizeof(Widget<int>));
}
```

Как при компиляции *b.cc* компилятор узнает, что должен использовать конкретный шаблон в `size(Widget<int>&)`, а не какой-то другой, например `size(Widget<double>&)`? Когда кто-то изменит *a.cc* и вернет `make_any (Widget<char>&)`, как компилятор узнает, что должен перекомпилировать *b.cc*, чтобы добавить экземпляр `size(Widget<char>&)` и удалить более не нужный `size(Widget<int>&)`, если, конечно, нет некоторого третьего файла *c.cc*, где требуется этот экземпляр?! То есть у компилятора нет никакой возможности узнать, какой код может потребоваться для чтения контейнера, который по определению способен хранить *любой* тип.

Поэтому, чтобы извлечь любую функцию значения в `any`, необходимо заранее знать тип содержащегося значения.

std::any и полиморфные типы

`std::any` занимает место между полиморфизмом времени компиляции типа `std::variant<A, B, C>` и полиморфизмом времени выполнения полиморфных иерархий наследования и `dynamic_cast`. Возможно, вам интересно узнать, взаимодействует ли `std::any` с механизмом `dynamic_cast`. Ответ: нет, и нет никакого стандартного способа получить это поведение. `std::any` на сто процентов обеспечивает статическую безопасность типа: его нельзя взломать и получить «указатель на данные» (например, `void *`), не зная точно тип хранящихся данных:

```
struct Animal {
  virtual ~Animal() = default;
};

struct Cat : Animal {};

void test()
{
  std::any a = Cat{};

  // Хранится объект "Cat"...
  assert(a.type() == typeid(Cat));
  assert(std::any_cast<Cat>(&a) != nullptr);

  // Попытка получить как базовый объект "Animal"
  // завершится неудачей.
  assert(a.type() != typeid(Animal));
  assert(std::any_cast<Animal>(&a) == nullptr);

  // Попытка получить void* тоже не увенчается успехом!
  assert(std::any_cast<void>(&a) == nullptr);
}
```

Коротко о стирании типа

Давайте теперь посмотрим, как тип `std::any` может быть реализован в стандартной библиотеке. Основная идея называется «стирание типа» и состоит в том, чтобы определить основные операции, которые должны поддерживаться для всех типов `T`, и затем «стереть» все остальные специфические операции, которые могут поддерживаться любым конкретным типом `T`.

Для `std::any` такими основными операциями являются:

- конструирование копии содержащегося объекта;
- конструирование копии содержащегося объекта «перемещением»;
- получение `typeid` содержащегося объекта.

Необходимы также конструирование и уничтожение, но эти две операции имеют отношение к управлению жизненным циклом самого содержащегося объекта, а не к списку «что можно с ним сделать», поэтому, по крайней мере в этом случае, мы не будем их рассматривать.

Итак, представим полиморфный класс (назовем его `AnyBase`), поддерживающий только эти три операции в виде виртуальных методов, которые можно переопределять, и затем будем создавать совершенно новый производный класс (назовем его `AnyImpl<T>`), когда программист фактически будет сохранять объект конкретного типа `T` в `any`:

```
class any;

struct AnyBase {
  virtual const std::type_info& type() = 0;
  virtual void copy_to(any&) = 0;
  virtual void move_to(any&) = 0;
  virtual ~AnyBase() = default;
};

template<typename T>
struct AnyImpl : AnyBase {
  T t_;
  const std::type_info& type() {
    return typeid(T);
  }
  void copy_to(any& rhs) override {
    rhs.emplace<T>(t_);
  }
  void move_to(any& rhs) override {
    rhs.emplace<T>(std::move(t_));
  }
  // в данном случае не требуется какой-то
  // особый деструктор
};
```

С этими вспомогательными классами код реализации `std::any` становится удивительно простым, особенно если для управления жизненным циклом нашего объекта `AnyImpl<T>` использовать умные указатели (см. главу 6 «Умные указатели»):

```
class any {
  std::unique_ptr<AnyBase> p_ = nullptr;
public:
  template<typename T, typename... Args>
  std::decay_t<T>& emplace(Args&&... args) {
    p_ = std::make_unique<AnyImpl<T>>(std::forward<Args>(args)...);
  }

  bool has_value() const noexcept {
    return (p_ != nullptr);
  }

  void reset() noexcept {
    p_ = nullptr;
  }

  const std::type_info& type() const {
    return p_ ? p_->type() : typeid(void);
  }

  any(const any& rhs) {
    *this = rhs;
  }

  any& operator=(const any& rhs) {
    if (rhs.has_value()) {
      rhs.p_->copy_to(*this);
    }
    return *this;
  }
};
```

В предыдущем примере кода отсутствует реализация присваивания перемещением. Это можно сделать так же, как присваивание копированием или просто поменяв местами указатели. Фактически стандартная библиотека предпочитает менять указатели местами всегда, когда это возможно, потому что это гарантирует отсутствие исключений; единственная причина, по которой `std::any` не меняет указатели, – использование «оптимизации маленьких объектов», чтобы избежать выделения динамической памяти для очень маленьких типов `T` с конструктором перемещения, не возбуждающим исключений. На момент написания этих строк libstdc++ (библиотека, используемая компилятором GCC) использует оптимизацию маленьких объектов и не выделяет память в куче для типов с размером до 8 байт; libc++ (библиотека, используемая

компилятором Clang) использует оптимизацию маленьких объектов для типов с размером до 24 байт.

В отличие от стандартных контейнеров, обсуждавшихся в главе 4 «Зоопарк контейнеров», `std::any` не принимает параметра с диспетчером памяти и не позволяет настраивать источник памяти. Если вы используете C++ в системе реального времени или с ограниченным объемом памяти, где размещение в динамической памяти не поддерживается, тогда вы не должны использовать тип `std::any`. Вместо него можно использовать, например, `tj::inplace_any<Size, Alignment>` из библиотеки Тимо Янга (Tiemo Jung). Если ничего другого не остается, вы теперь знаете, как написать свою реализацию!

std::any и копирование

Обратите внимание: наша реализация `AnyImpl<T>::copy_to` требует, чтобы тип `T` имел конструктор копирования. Это также верно для стандартного типа `std::any`; просто нет способа сохранить в объекте `std::any` значение, обладающее только конструктором перемещения. Обойти это ограничение можно, добавив «тонкую» обертку, обеспечивающую соответствие объекта, поддерживающего только перемещение, синтаксическому требованию о конструкторе копирования, без фактического копирования:

```
using Ptr = std::unique_ptr<int>;

template<class T>
struct Shim {
  T get() { return std::move(*t_); }

  template<class... Args>
  Shim(Args&&... args) : t_(std::in_place,
    std::forward<Args>(args)...) {}

  Shim(Shim&&) = default;
  Shim& operator=(Shim&&) = default;
  Shim(const Shim&) { throw "oops"; }
  Shim& operator=(const Shim&) { throw "oops"; }
private:
  std::optional<T> t_;
};

void test()
{
  Ptr p = std::make_unique<int>(42);

  // Ptr нельзя сохранить в std::any, потому что доступен только для перемещения.
  // std::any a = std::move(p);

  // А Shim<Ptr> можно!
  std::any a = Shim<Ptr>(std::move(p));
```

```
  assert(a.type() == typeid(Shim<Ptr>));

  // Перемещение Shim<Ptr> выполняется нормально...
  std::any b = std::move(a);

  try {
    // ...но попытка скопировать Shim<Ptr> вызывает исключение.
    std::any c = b;
  } catch (...) {}

  // Получить Ptr из Shim<Ptr>.
  Ptr r = std::any_cast<Shim<Ptr>&>(b).get();
  assert(*r == 42);
}
```

Обратите внимание на использование `std::optional<T>` в предыдущем примере; он защищает наш поддельный конструктор копирования от ситуации, когда `T` не имеет конструктора по умолчанию.

И снова о стирании типов: std::function

Теперь мы знаем, что основными операциями для `std::any` являются:

- конструирование копии содержащегося объекта;
- конструирование копии содержащегося объекта «перемещением»;
- получение `typeid` содержащегося объекта.

Допустим, мы решили добавить к этому набору еще одну операцию, чтобы он включал:

- конструирование копии содержащегося объекта;
- конструирование копии содержащегося объекта «перемещением»;
- получение `typeid` содержащегося объекта;
- вызов содержащегося объекта с фиксированной последовательностью аргументов определенных типов `A...` и преобразование результата в некоторый фиксированный тип `R`.

Стирание типа этого набора операций соответствует стандартному библиотечному типу `std::function<R(A...)>`!

```
int my_abs(int x) { return x < 0 ? -x : x; }
long unusual(long x, int y = 3) { return x + y; }

void test()
{
  std::function<int(int)> f; // сконструировать пустой контейнер
  assert(!f);

  f = my_abs; // сохранить функцию в контейнер
```

```
  assert(f(-42) == 42);

  f = [](long x) { return unusual(x); }; // или лямбда-выражение!
  assert(f(-42) == -39);
}
```

Копирование `std::function` всегда влечет копирование содержащегося объекта, если содержащийся объект имеет состояние. Конечно, если этот содержащийся объект является указателем на функцию, вы не заметите никакой разницы; но факт копирования можно заметить, если использовать объект пользовательского типа или лямбда-выражение с состоянием:

```
f = [i=0](int) mutable { return ++i; };
assert(f(-42) == 1);
assert(f(-42) == 2);

auto g = f;
assert(f(-42) == 3);
assert(f(-42) == 4);
assert(g(-42) == 3);
assert(g(-42) == 4);
```

Так же как в случае с `std::any`, `std::function<R(A...)>` позволяет извлечь `typeid` содержащегося объекта или указатель на сам объект, если его можно определить статически:

- `f.target_type()` эквивалентно `a.type()`;
- `f.target<T>()` эквивалентно `std::any_cast<T*>(&a)`.

```
if (f.target_type() == typeid(int(*)(int))) {
  int (*p)(int) = *f.target<int (*)(int)>();
  use(p);
} else {
  // обработка ошибки!
}
```

Замечу при этом, что в своей практике я не видел случаев применения этих методов. Обычно, если возникает необходимость узнать тип объекта, содержащегося в `std::function`, это говорит о том, что вы сделали что-то неправильно.

Главное предназначение `std::function` – служить словарным типом для передачи «поведения» через границы модулей, когда использование шаблонов оказывается невозможно, например когда нужно передать обратный вызов в функцию из сторонней библиотеки или когда вы пишете библиотеку, которая должна принимать обратные вызовы от вызывающего кода:

```
// templated_for_each - это шаблон и должен быть
// видим в точке вызова.
template<class F>
void templated_for_each(std::vector<int>& v, F f) {
```

```
  for (int& i : v) {
    f(i);
  }
}

// type_erased_for_each имеет постоянный ABI и фиксированный адрес.
// Может вызываться, только если объявление находится в области видимости.
extern void type_erased_for_each(std::vector<int>&,
  std::function<void(int)>);
```

Мы начали эту главу с разговора о `std::string`, стандартном словарном типе для передачи строк между функциями; теперь, приблизившись к концу главы, мы занялись обсуждением `std::function`, стандартного словарного типа для передачи функций в функции!

std::function, копирование и размещение в динамической памяти

Так же как `std::any`, `std::function` требует, чтобы любой объект, сохраняемый в объекте этого типа, имел конструктор копирования. Это может стать проблемой при использовании лямбда-выражений, хранящих `std::future<T>`, `std::unique_ptr<T>` или другие типы, поддерживающие только перемещение: такие лямбда-выражения сами будут доступны лишь для перемещения. Одно из решений данной проблемы было представлено в разделе «std::any и копирование» выше: мы можем определить обертку, которая синтаксически поддерживает копирование, но возбуждает исключение при попытке скопировать ее.

При работе с `std::function` и замыканиями в лямбда-выражениях часто предпочтительнее замкнуть лямбда-выражение, доступное только для перемещения, в `shared_ptr`. Подробнее о типе `shared_ptr` рассказывается в следующей главе:

```
auto capture = [](auto& p) {
  using T = std::decay_t<decltype(p)>;
  return std::make_shared<T>(std::move(p));
};

std::promise<int> p;

std::function<void()> f = [sp = capture(p)]() {
  sp->set_value(42);
};
```

Так же как `std::any`, `std::function` *не* имеет параметра для передачи диспетчера памяти и не позволяет настраивать или изменять источник динамической памяти. Если вы используете C++ в системе реального времени или с ограниченным объемом памяти, где размещение в динамической памяти не поддерживается, тогда вы не должны использовать тип `std::function`. Вмес-

то него можно использовать, например, `sg14::inplace_function<R(A...), Size, Alignment>` из библиотеки Карла Кука (Carl Cook).

Итоги

Словарные типы, такие как `std::string` и `std::function`, образуют основу для работы с общими программными понятиями. В C++17 имеется богатый набор словарных типов для представления *алгебраических данных*: `std::pair` и `std::tuple` (типы-произведения), `std::optional` и `std::variant` (типы-суммы) и `std::any` (всеобъемлющий тип-сумма – он может хранить почти все, что угодно). Но не увлекайтесь и не пытайтесь начать возвращать `std::tuple` и `std::variant` из каждой функции! Именованные классы по-прежнему являются эффективным способом сохранения удобочитаемости кода.

Используйте `std::optional` как сигнал о возможной утечке значения или «пока отсутствующем» члене данных.

Используйте `std::get_if<T>(&v)` для определения типа варианта; используйте `std::any_cast<T>(&a)`, чтобы узнать тип объекта в `any`. Помните, что тип в параметре должен точно совпадать с типом содержащегося объекта, иначе вы получите `nullptr`.

Не забывайте, что `make_tuple` и `make_pair` не только конструируют кортежи и пары объектов; они также преобразуют объекты `reference_wrapper` в обычные ссылки. Используйте `std::tie` и `std::forward_as_tuple` для создания кортежей ссылок. `std::tie` очень удобно применять для множественного присваивания и для записи операторов сравнения. `std::forward_as_tuple` может пригодиться для метапрограммирования.

Имейте в виду, что `std::variant` может оказаться в состоянии «без значения из-за исключения» (valueless by exception); но это не представляет никакой проблемы, если вы не пишете классы, конструкторы перемещения которых могут возбуждать исключение. Замечу особо: не пишите классы с конструкторами перемещения, которые могут возбуждать исключение!

Помните, что типы со *стиранием типов* `std::any` и `std::function` неявно используют динамическую память. Сторонние библиотеки предоставляют нестандартные `inplace_`-версии этих типов. Типы `std::any` и `std::function` требуют поддержки копирования от содержащихся в них типов. Используйте «замыкание в `shared_ptr`» в таких случаях.

Глава 6

Умные указатели

C++ удерживает передовые позиции в программной индустрии благодаря своей производительности – хорошо написанный код на C++ *почти* по определению выполняется быстрее, чем код на любом другом языке, потому что C++ дает программисту почти полный контроль над кодом, который в конечном счете генерирует компилятор.

Одной из классических особенностей низкоуровневого, высокопроизводительного кода является использование *простых указателей* (`Foo*`). Однако такие указатели имеют массу ловушек, таких как утечки памяти и недействительные указатели. Типы «умных указателей» в библиотеке C++11 способны помочь избежать этих ловушек при минимальных затратах.

В этой главе вы:

- познакомитесь с определением «умный указатель» и узнаете, как писать свои умные указатели;
- увидите, как `std::unique_ptr<T>` помогает предотвратить утечки ресурсов любых видов (не только памяти);
- узнаете, как реализован тип `std::shared_ptr<T>` и как он влияет на использование памяти;
- познакомитесь с шаблоном проектирования «Странно рекурсивный шаблон» (Curiously Recurring Template Pattern).

История появления умных указателей

Простые указатели широко использовались в языке C:

- как недорогое, некопируемое представление объекта, которым владеет вызывающий код;
- как способ передачи объектов в функции для их изменения;
- как половина пары указатель/длина, используемой для представления массивов;
- как необязательный аргумент (который может быть действительным указателем или `null`);
- как средство управления памятью в куче.

Для поддержки первых двух пунктов из этого списка в C++ имеются обычные ссылки (`const Foo&` и `Foo&`); плюс в большинстве случаев семантика перемещения удешевляет получение и возврат сложных объектов по значению. Для решения некоторых задач, соответствующих первому и третьему пунктам, в C++17 можно использовать `std::string_view`. И мы только что видели в главе 5 «Словарные типы», что `optional<T>` – и иногда `optional<reference_wrapper<T>>` – позволяет реализовать четвертый пункт.

В этой главе мы сосредоточимся на пятом пункте.

Распределение памяти из кучи в C сопровождается множеством проблем, и все эти проблемы (и многие другие!) проявляются в версиях C++ до 2011. Однако с выходом C++11 большинство проблем было решено. Перечислим их.

- **Утечки памяти:** можно выделить память из кучи и по ошибке забыть написать код, освобождающий ее.
- **Утечки памяти:** можно написать код, освобождающий память, но из-за исключения или преждевременного возврата он не будет выполнен, и память останется занятой!
- **Использование памяти после освобождения:** можно скопировать указатель на объект в куче и затем освободить память, используя оригинальный указатель. В этом случае владелец копии указателя не заметит, что указатель стал недействительным.
- **Повреждение кучи выполнением арифметических операций с указателями:** можно разместить массив в куче по адресу *A*. Наличие простого указателя на массив вынуждает использовать арифметику указателей, и в конце вы можете по ошибке освободить память, используя указатель с адресом *A* + *k*. Когда *k = 0* (закон Мерфи гарантирует, что при тестировании так и будет), это не вызовет никаких проблем; но при *k=1* вы повредите кучу и вызовете аварийное завершение программы.

Первые две проблемы усугубляются тем фактом, что семантически попытка выделения памяти может завершиться неудачей – `malloc` может вернуть `null`, оператор `new` может возбудить исключение `std::bad_alloc` – а это означает, что в версиях языка до C++11 приходится писать массу кода, обрабатывающего неудачное развитие событий. (В C++ вы всегда «пишете» этот код, подозревая об этом или нет, потому что пути обработки исключений существуют, даже если не думаете о них осознанно.) Итогом всего этого является высокая сложность управления динамической памятью в C++.

Если не использовать умные указатели!

Умные указатели никогда ничего не забывают

Идея «умного указателя» (не путайте с «причудливыми указателями» (fancy pointer), о которых рассказывается в главе 8 «Диспетчеры памяти») заключается в том, что это класс – обычно шаблонный, – который синтаксически действует как указатель, но имеет специальные функции-члены (конструктор,

деструктор и копирования/перемещения), выполняющие операции, необходимые для обеспечения безопасности. Например:

- деструктор такого указателя также освобождает память, на которую он указывает, помогая решить проблему утечки памяти;
- возможно, указатель не разрешает копировать себя, помогая решить проблему использования памяти после освобождения;
- или указатель разрешает копирование, но знает, сколько копий существует, и не освобождает память, пока не будет уничтожена последняя копия;
- или указатель разрешает копирование и освобождает память при уничтожении, но после этого все его копии, как по волшебству, получают значение `null`;
- или указатель не поддерживает встроенный `operator+`, помогая решить проблему повреждения кучи из-за применения арифметики указателей;
- или указатель поддерживает арифметические операции, но арифметика «объект, на который ссылается указатель» обрабатывается отдельно от арифметики «объект, который должен освобождаться».

К стандартным типам умных указателей относятся: `std::unique_ptr<T>`, `std::shared_ptr<T>` и (в действительности не совсем указатель, но мы будем использовать его вместе с указателями) `std::weak_ptr<T>`. В этой главе мы подробно рассмотрим каждый из этих трех типов, а также один нестандартный, но очень полезный тип, который может стать стандартным в будущем!

Автоматическое управление памятью с std::unique_ptr<T>

Умные указатели имеют простой набор основных свойств: они должны поддерживать `operator*` и перегружать специальные функции-члены для поддержки любых ограничений (инвариантов) класса.

`std::unique_ptr<T>` реализует тот же интерфейс, что и `T*`, но с одним ограничением: после создания экземпляра `unique_ptr`, указывающего на объект в куче, этот объект автоматически освобождается при вызове деструктора `unique_ptr`. Давайте напишем код, который поддерживал бы интерфейс `T*`:

```
template<typename T>
class unique_ptr {
  T *m_ptr = nullptr;
public:
  constexpr unique_ptr() noexcept = default;
  constexpr unique_ptr(T *p) noexcept : m_ptr(p) {}

  T *get() const noexcept { return m_ptr; }
  operator bool() const noexcept { return bool(get()); }
```

```
T& operator*() const noexcept { return *get(); }
T* operator->() const noexcept { return get(); }
```

Если остановиться на этом – просто дав способ сконструировать объект указателя из T* и получить указатель обратно, – мы получим observer_ptr<T>, который обсуждается в конце этой главы. Но давайте продолжим. Добавим методы release и reset:

```
void reset(T *p = nullptr) noexcept {
  T *old_p = std::exchange(m_ptr, p);
  delete old_p;
}

T *release() noexcept {
  return std::exchange(m_ptr, nullptr);
}
```

p.release() действует подобно p.get(), но, кроме возврата копии оригинального указателя, он также записывает null в p (не освобождая оригинальный указатель, потому что вызывающий код обращением к этому методу принимает на себя ответственность за владение указателем).

p.reset(q) освобождает текущее содержимое p и на его место помещает новый указатель q.

Обратите внимание, что мы реализовали обе эти функции-члены в терминах стандартного алгоритма std::exchange, о котором не рассказывалось в главе 3 «Алгоритмы с парами итераторов». Это разновидность std::swap: она сохраняет новое значение и возвращает прежнее.

Наконец, на основе этих двух простых операций можно реализовать специальные функции-члены типа std::unique_ptr<T> для поддержки наших ограничений: после получения указателя объектом unique_ptr он остается действительным, пока объект unique_ptr имеет то же самое значение, а когда это не так, то есть когда в unique_ptr записывается другой адрес или он уничтожается, исходный указатель освобождается. Вот эти функции-члены:

```
unique_ptr(unique_ptr&& rhs) noexcept {
  this->reset(rhs.release());
}

unique_ptr& operator=(unique_ptr&& rhs) noexcept {
  reset(rhs.release());
  return *this;
}

~unique_ptr() {
  reset();
}
};
```

На рис. 6.1 показано, как хранится в памяти наш std::unique_ptr<T>.

```
std::unique_ptr<T> p;
```

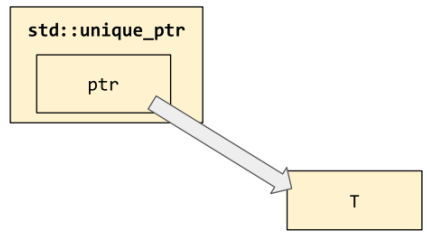


Рис. 6.1. Размещение в памяти объекта типа std::unique_ptr<T>

Нам понадобится еще одна вспомогательная функция, которая поможет никогда не пачкать руки о низкоуровневые указатели:

```
template<typename T, typename... Args>
unique_ptr<T> make_unique(Args&&... args)
{
  return unique_ptr<T>(new T(std::forward<Args>(args)...));
}
```

Имея `unique_ptr` в своем арсенале, можно заменить устаревший код, например такой:

```
struct Widget {
  virtual ~Widget();
};
struct WidgetImpl : Widget {
  WidgetImpl(int size);
};
struct WidgetHolder {
  void take_ownership_of(Widget *) noexcept;
};
void use(WidgetHolder&);

void test() {
  Widget *w = new WidgetImpl(30);
  WidgetHolder *wh;
  try {
    wh = new WidgetHolder();
  } catch (...) {
    delete w;
    throw;
  }
  wh->take_ownership_of(w);
  try {
    use(*wh);
  } catch (...) {
```

```
    delete wh;
    throw;
  }
  delete wh;
}
```

более современным кодом в стиле C++17:

```
void test() {
  auto w = std::make_unique<WidgetImpl>(30);
  auto wh = std::make_unique<WidgetHolder>();
  wh->take_ownership_of(w.release());
  use(*wh);
}
```

Обратите внимание, что unique_ptr<T> является еще одним применением идиомы RAII – в данном случае буквально. Хотя «интересное» действие (освобождение внутреннего указателя) все еще происходит в ходе уничтожения (unique_ptr), единственный способ получить максимальную выгоду от unique_ptr – при каждом выделении ресурса инициализировать unique_ptr для управления им. Функция std::make_unique<T>(), представленная в предыдущем разделе (и появившаяся в C++14), – это ключевой аспект безопасного управления памятью в современном C++.

Хотя unique_ptr можно использовать без make_unique, но поступать так нежелательно:

```
std::unique_ptr<Widget> bad(new WidgetImpl(30));
bad.reset(new WidgetImpl(40));

std::unique_ptr<Widget> good = std::make_unique<WidgetImpl>(30);
good = std::make_unique<WidgetImpl>(40);
```

Почему в C++ нет ключевого слова finally

Рассмотрим еще раз фрагмент «устаревшего» кода из предыдущего раздела:

```
try {
  use(*wh);
} catch (...) {
  delete wh;
  throw;
}
delete wh;
```

В других языках программирования, таких как In Java и Python, эту семантику можно выразить более компактно, воспользовавшись ключевым словом finally:

```
try {
  use(*wh);
```

```
} finally {
  delete wh;
}
```

В C++ нет ключевого слова `finally`, и ничего не говорит о том, что оно будет добавлено в язык. Это обусловлено отличием философии C++ от других языков. В C++, если вы заинтересованы в принудительном применении некоторого ограничения, например: «этот указатель всегда будет освобождаться в конце блока, независимо от того, как мы его достигнем», – вам не придется писать явный код, потому что иначе появляется шанс написать его неправильно и породить ошибки.

Если есть какие-то ограничения, которые вы хотели бы гарантировать, лучшим местом для этого является *система типов*, использование конструкторов, деструкторов и других специальных функций-членов – инструментов RAII. С их помощью вы сможете гарантировать, что при *любом возможном* использовании ваш новый тип обеспечит соблюдение ограничений, таких как «внутренний указатель всегда должен освобождаться, когда больше не удерживается объектом этого типа». И когда вы приступите к реализации бизнес-логики, вам не придется соблюдать ограничения явно, а ваш код будет выглядеть простым и при этом всегда иметь правильное поведение.

То есть если вы поймали себя на том, что пишете или даже просто хотите написать код, напоминающий предыдущий пример с `finally`, остановитесь и подумайте: «подходит ли `unique_ptr` для данного случая» или «должен ли я писать класс RAII для этой цели».

Настройка обратного вызова удаления

Коль скоро речь зашла о нестандартных типах RAII, вам может быть интересно узнать, можно ли использовать `std::unique_ptr` с собственным обратным вызовом удаления: например, вместо передачи внутреннего указателя оператору `delete` у вас может появиться желание передать его в вызов `free()`. Да, это возможно!

`std::unique_ptr<T,D>` имеет второй параметр типа: *тип обратного вызова удаления*. Параметр `D` по умолчанию принимает значение `std::default_delete<T>`, которое просто вызывает оператор `delete`, но вы можете передать любой желаемый тип – обычно класс, определяемый пользователем с перегруженным `operator()`:

```
struct fcloser {
  void operator()(FILE *fp) const {
    fclose(fp);
  }

  static auto open(const char *name, const char *mode) {
    return std::unique_ptr<FILE, fcloser>(fopen(name, mode));
  }
```

```
};

void use(FILE *);

void test() {
  auto f = fcloser::open("test.txt", "r");
  use(f.get());
  // f будет закрыт, даже если use() возбудит исключение
}
```

Кстати, обратите внимание: деструктор `std::unique_ptr` написан так, что гарантированно не вызовет ваш обратный вызов с пустым указателем. Это особенно важно для предыдущего примера, потому что `fclose(NULL)` – это особый случай, означающий: «закрыть все дескрипторы файлов, открытые текущим процессом», который почти всегда далек от желаемого!

Обратите также внимание, что `std::make_unique<T>()` принимает только один параметр типа – не существует версии `std::make_unique<T,D>()`. Но правило, предлагающее не касаться руками простых указателей, продолжает действовать и в этом случае; именно поэтому вызов `fopen` и создание `unique_ptr` в предыдущем примере завернуто в удобную вспомогательную функцию `fcloser::open` вместо встраивания вызова `fopen` в тело `test`.

Пространство для вашего обратного вызова удаления будет выделено в теле самого объекта `std::unique_ptr<T,D>`, то есть `sizeof(unique_ptr<T,D>)` может вернуть большее значение, чем `sizeof(unique_ptr<T>)`, если `D` имеет данные-члены (см. рис. 6.2).

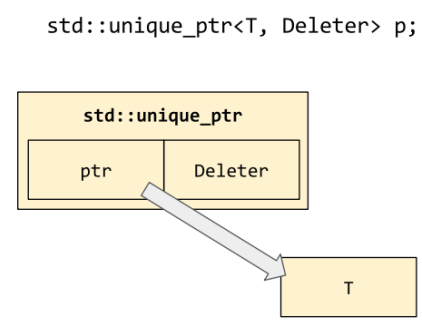


Рис. 6.2. Размещение в памяти unique_ptr<T,D>

Управление массивами с помощью std::unique_ptr<T[]>

Другой случай, когда `delete p` оказывается не лучшим способом освобождения простого указателя, – если `p` указывает на первый элемент массива; в этом случае следует использовать `delete [] p`. К счастью, в C++14 появился тип `std::unique_ptr<T[]>`, и он действует правильно в этой ситуации (в силу

того факта, что также существует `std::default_delete<T[]>` и вызывает оператор `delete[]`).

Для массивов существует перегруженная версия `std::make_unique`, но будьте внимательны – ее аргументы имеют другой смысл! `std::make_unique<T[]>(n)`, по сути, соответствует вызову `new T[n]()`, где круглые скобки означают, что она выполняет инициализацию значений всех элементов; то есть обнулять элементарные типы. В редких случаях, когда такое поведение нежелательно, вам придется вручную вызвать `new` и завернуть возвращаемое значение в `std::unique_ptr<T[]>`. Делать это предпочтительнее с применением вспомогательной функции, как в примере из предыдущего раздела (где использовалась функция `fcloser::open`).

Подсчет ссылок с std::shared_ptr<T>

Полностью избавившись от проблемы утечки памяти, перейдем теперь к проблеме «использования после освобождения». Существенной составляющей этой проблемы, которую нужно устранить, является *неясность владения* или, скорее, *совместное владение* данным ресурсом или блоком памяти. В такой блок памяти могут «заглядывать» разные участки кода, в разные времена и, возможно, посредством разных структур данных или из разных потоков выполнения, и нам нужно гарантировать вовлеченность всех заинтересованных сторон в принятие решения о моменте, когда этот блок памяти можно освободить. Владение блоком памяти должно быть по-настоящему *совместным*.

Для этой цели стандарт предлагает тип `std::shared_ptr<T>`. Внешне его интерфейс очень напоминает `std::unique_ptr<T>`; все отличия скрыты «под капотом», внутри реализаций специальных функций-членов.

`std::shared_ptr<T>` реализует подход к управлению памятью, который широко известен как *подсчет ссылок*. Каждый объект, управляемый посредством `shared_ptr`, хранит счетчик ссылок на него, то есть количество сторон, интересующихся этим объектом в данный момент, и как только значение счетчика уменьшается до нуля, объект понимает, что настал момент освободить занимаемую им память. На самом деле, конечно, речь идет не об объекте, который «освобождает себя сам», а о тонкой обертке – «управляющем блоке», – умеющей подсчитывать ссылки и освобождать фактические объекты, которая автоматически создается в куче всякий раз, когда вы передаете право владения экземпляру `shared_ptr`. Управляющий блок обслуживается незаметно, самой библиотекой, но если заглянуть в память, вы увидите картину, изображенную на рис. 6.3.

По аналогии с парой `unique_ptr`/`make_unique`, стандартная библиотека сопровождает `shared_ptr` функцией `make_shared`, чтобы избавить вас от необходимости пачкать руки об обычные указатели. Другое преимущество использования `std::make_shared<T>(args)` для размещения общих объектов состоит в том, что передача владения экземпляру `shared_ptr` требует выделения дополнительной памяти для управляющего блока. Вызывая `make_shared`, вы раз-

решаете библиотеке выделить единый блок памяти достаточного размера для хранения управляющего блока и вашего объекта T. (Это иллюстрируют прямоугольники control_block_impl и Super на рис. 6.3.)

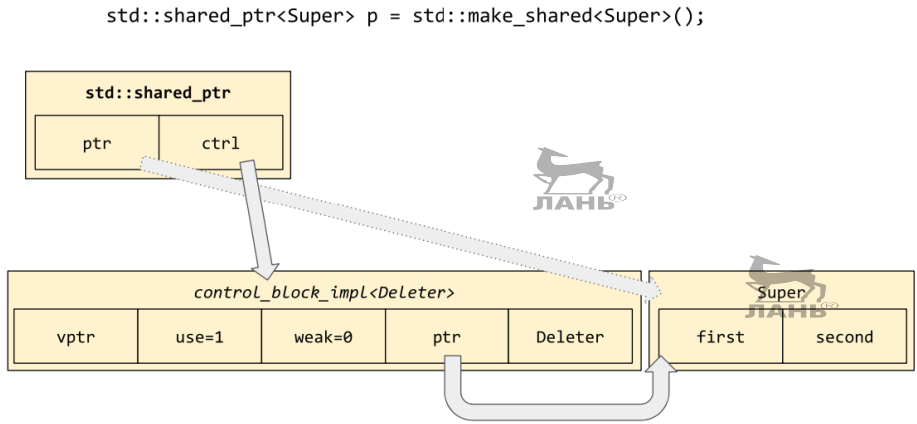


Рис. 6.3. Организация std::shared_ptr<T> в памяти

Операция копирования shared_ptr увеличивает счетчик use в управляющем блоке; операция освобождения shared_ptr – уменьшает его. Операция присваивания указателю shared_ptr уменьшит счетчик use прежнего значения (если имеется) и увеличит счетчик use нового значения. Вот несколько примеров использования shared_ptr:

```
std::shared_ptr<X> pa, pb, pc;

pa = std::make_shared<X>();
// счетчик use всегда получает начальное значение 1

pb = pa;
// копирование указателя; теперь счетчик use равен 2

pc = std::move(pa);
assert(pa == nullptr);
// перемещение указателя сохраняет 2 в счетчике use

pb = nullptr;
// уменьшит use до 1
assert(pc.use_count() == 1);
```

Диаграмма на рис. 6.4 иллюстрирует интересный и иногда полезный аспект shared_ptr: возможность для двух экземпляров shared_ptr ссылаться на общий управляющий блок и указывать на разные участки памяти, которая контролируется этим управляющим блоком.

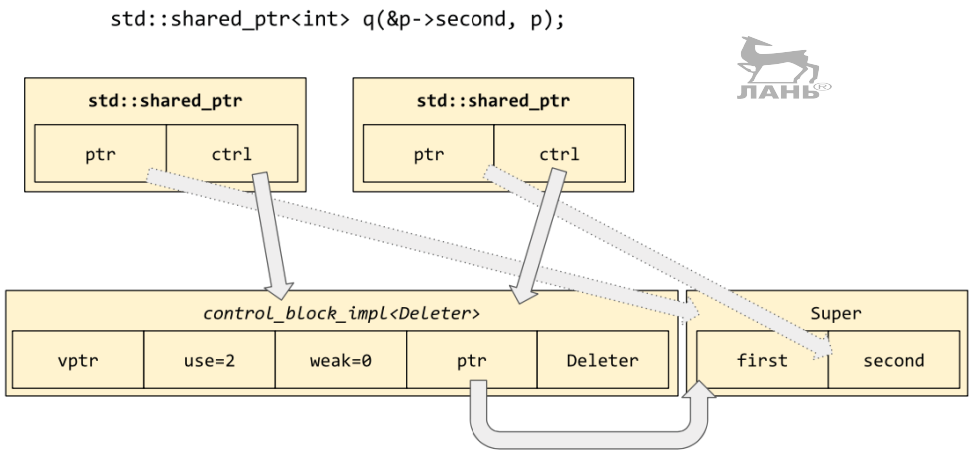


Рис. 6.4. Два экземпляра `shared_ptr` ссылаются на общий управляющий блок и указывают на разные участки памяти

Конструктор, использованный в диаграмме на рис. 6.4, который также используется в функции `get_second()`, часто называют «конструктором псевдонима» для `shared_ptr`. Он принимает существующий объект непустого указателя `shared_ptr` любого типа, управляющий блок которого требуется совместно использовать с вновь созданным объектом. Следующий пример кода выведет сообщение `"destroying Super"` только после сообщения `"accessing Super::second"`:

```
struct Super {
  int first, second;
  Super(int a, int b) : first(a), second(b) {}
  ~Super() { puts("destroying Super"); }
};

auto get_second() {
  auto p = std::make_shared<Super>(4, 2);
  return std::shared_ptr<int>(p, &p->second);
}

void test() {
  std::shared_ptr<int> q = get_second();
  puts("accessing Super::second");
  assert(*q == 2);
}
```

Как видите, после передачи владения системе `shared_ptr` ответственность за запоминание, когда должен освобождаться ресурс, полностью перекладывается на управляющий блок. Но совсем необязательно, чтобы ваш код использовал `shared_ptr<T>` только потому, что управляемый объект имеет тип `T`.

Не допускайте двойного управления!

Тип shared_ptr<T> способен избавить ваш код от неприятных ошибок повторного освобождения ресурсов, но, к сожалению, неопытные программисты слишком часто создают такие ошибки с shared_ptr, злоупотребляя конструкторами, которые принимают простые указатели. Например:

```
std::shared_ptr<X> pa, pb, pc;

pa = std::make_shared<X>();
  // счетчик use всегда получает начальное значение 1

pb = pa;
  // копирование указателя; теперь счетчик use равен 2

pc = std::shared_ptr<X>(pb.get()); // ОШИБКА!
  // передача того же указателя в shared_ptr снова,
  // из-за чего возникает двойное управление!
assert(pb.use_count() == 2);
assert(pc.use_count() == 1);

pc = nullptr;
  // счетчик use в pc достигает нуля и shared_ptr
  // вызывает "delete" для объекта X
*pb; // поведение обращения к уничтоженному объекту непредсказуемо
```

Не забывайте, что вы не должны касаться своими руками простых указателей! Место, где в этом коде допущена первая ошибка, – вызов pb.get() для получения обычного указателя из shared_ptr.

В данном случае ошибку можно исправить вызовом конструктора псевдонима, pc = std::shared_ptr<X>(pb, pb.get()), но такой вызов имеет тот же эффект, как и простое присваивание pc = pb. Отсюда вытекает еще одно общее правило: если вам приходится явно использовать слово shared_ptr в своем коде, значит, вы делаете что-то необычное и, может быть, опасное. Вы уже можете выделять память, управлять объектами в куче (с помощью std::make_shared) и манипулировать счетчиком use управляемых объектов, создавая и уничтожая копии указателя (применяя auto для объявления необходимых переменных), и все это без обращения к shared_ptr. Единственное место, где это правило не действует, – когда требуется объявить член данных класса типа shared_ptr<T> – это невозможно сделать без упоминания имени типа!

Удерживание обнуляемых дескрипторов с помощью weak_ptr

Возможно, на предыдущих диаграммах вы заметили странный элемент weak. Теперь пришло время рассказать о нем.

Иногда – хоть и редко – бывает желательно использовать shared_ptr для управления владением общими объектами и хранить указатель на объект, не

выражая владение этим объектом. Конечно, для выражения идеи «невладеющей ссылки» можно использовать обычный указатель, ссылку или `observer_ptr<T>`, но тогда возникает опасность, что фактический владелец объекта решит освободить его, а мы после этого попытаемся разыменовать наш невладеющий указатель, чтобы прочитать уничтоженный объект. Это опасное поведение иллюстрирует `DangerousWatcher` в следующем примере:

```
struct DangerousWatcher {
  int *m_ptr = nullptr;

  void watch(const std::shared_ptr<int>& p) {
    m_ptr = p.get();
  }

  int current_value() const {
    // В этот момент *m_ptr может быть уничтожен!
    return *m_ptr;
  }
};
```

Для выражения «ссылки» можно было бы использовать `shared_ptr`, но в результате мы получим владеющую ссылку и из наблюдателя (Watcher) превратимся в соучастника (Participant):

```
struct NotReallyAWatcher {
  std::shared_ptr<int> m_ptr;

  void watch(const std::shared_ptr<int>& p) {
    m_ptr = p;
  }

  int current_value() const {
    // Теперь *m_ptr не может быть уничтожен; наше
    // соучастие помешает этому!
    return *m_ptr;
  }
};
```

В действительности же нам нужна невладеющая ссылка, которая тем не менее знакома с системой `shared_ptr` управления памятью и может обращаться к управляющему блоку для проверки существования целевого объекта. Но между моментом, когда мы узнали, что объект существует, и моментом, когда мы попытались обратиться к нему, он может быть уничтожен другим потоком выполнения! Поэтому нам нужна элементарная операция, «атомарно получающая владеющую ссылку (`shared_ptr`) на целевой объект, если он существует, или возвращающая отказ». То есть нам не нужна *невладеющая ссылка*; нам нужен *билет, который в будущем можно обменять на владеющую ссылку*.

В стандартной библиотеке есть такой «билет на `shared_ptr`», и называется он `std::weak_ptr<T>`. (Слово «weak» (слабая) в имени подчеркивает отличие

от «строгой» владеющей ссылки в shared_ptr.) Вот как можно использовать weak_ptr, чтобы решить нашу проблему наблюдателя Watcher:

```
struct CorrectWatcher {
  std::weak_ptr<int> m_ptr;

  void watch(const std::shared_ptr<int>& p) {
    m_ptr = std::weak_ptr<int>(p);
  }

  int current_value() const {
    // Теперь можно безопасно узнать, освобожден
    // ли *m_ptr или нет.
    if (auto p = m_ptr.lock()) {
      return *p;
    } else {
      throw "It has no value; it's been deallocated!";
    }
  }
};
```

Для практического применения weak_ptr достаточно знания всего двух операций: конструирования weak_ptr<T> из shared_ptr<T> (вызовом конструктора, как показано в функции watch()) и shared_ptr<T> из weak_ptr<T> вызовом wptr.lock(). Если объект был уничтожен к этому моменту, wptr.lock() вернет пустой указатель shared_ptr.

Проверить присутствие целевого объекта в памяти можно также с помощью функции-члена wptr.expired(), но она почти не имеет практической пользы, потому что даже если она в данный момент вернула false (объект существует), через несколько микросекунд она может вернуть true (объект уничтожен).

Диаграмма на рис. 6.5 продолжает диаграмму на рис. 6.4, добавляя создание weak_ptr из q и затем обнуляя исходный shared_ptr.

Операция копирования weak_ptr увеличивает счетчик weak в соответствующем управляющем блоке, а уничтожение weak_ptr – уменьшает его. Когда счетчик use достигает нуля, система понимает, что может без опаски удалить управляемый объект; но управляющий блок останется на месте, пока не обнулится счетчик weak, после чего его тоже можно освободить, потому что в системе не осталось ни одного объекта weak_ptr, ссылающегося на него (см. рис. 6.6).

Возможно, вы обратили внимание, что shared_ptr использует тот же трюк с Deleter, который мы видели в контексте std::any и std::function в главе 5 «Словарные типы», – *стирание типов*. И так же, как std::any и std::function, std::shared_ptr предоставляет функцию std::get_deleter<Deleter>(p) для получения оригинального объекта, выполняющего удаление. Однако этот лакомый кусочек совершенно бесполезен на практике. Я упомянул о нем, только чтобы обратить внимание на важность стирания типа в современном C++. Даже shared_ptr, чья цель как будто не имеет ничего общего со стиранием типов, полагается на стирание типа в части своих возможностей.

```
std::weak_ptr<int> w = q;
q = nullptr;
```

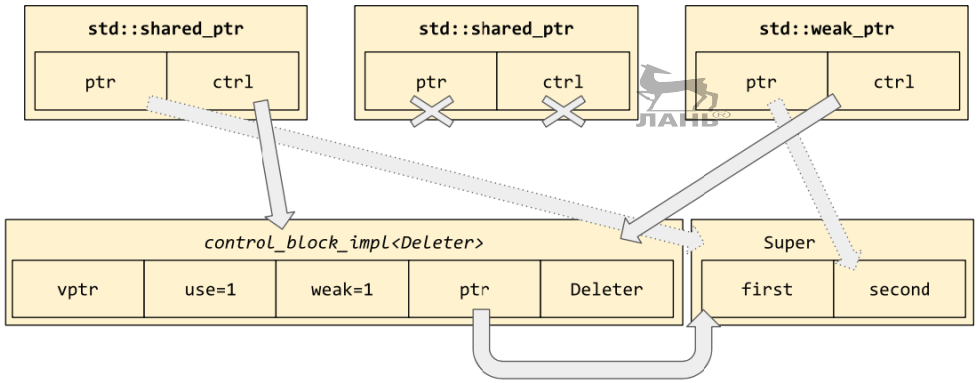


Рис. 6.5. Создание невладеющей ссылки и обнуление второго владеющего указателя shared_ptr

```
p = nullptr;  // уменьшит счетчик use до нуля
```

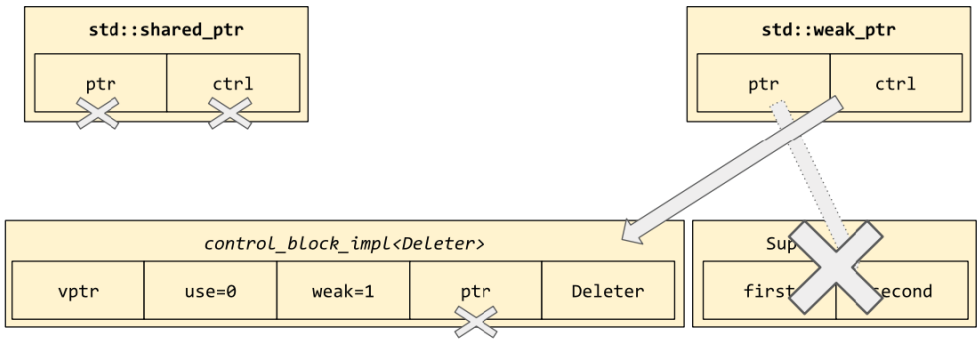


Рис. 6.6. Управляющий остается на месте, пока не обнулится счетчик weak

Сообщение информации о себе с std::enable_shared_from_this

Мы должны рассмотреть еще один аспект экосистемы `shared_ptr`. Выше уже упоминалось об опасности «двойного управления» созданием нескольких управляющих блоков. Соответственно, нам может понадобиться возможность узнать по указателю на объект в динамической памяти, кто управляет им в данный момент.

Это может пригодиться в объектно-ориентированном программировании, когда метод `A::foo()` желает вызвать некоторую внешнюю функцию `bar()`, а

функции bar() требуется обратный указатель на объект A. Если не задумываться о проблемах, связанных с жизненным циклом, сделать это легко; A::foo() мог бы просто выполнить вызов bar(this). А теперь представьте, что объект A управляется указателем shared_ptr и bar() внутри сохраняет копию указателя this – возможно, с целью регистрации обратного вызова для использования в будущем, или, может быть, мы запускаем параллельный поток выполнения, а A::foo() завершается и возвращает управление вызывающему коду. То есть нам нужен какой-то механизм, который не позволит уничтожить A, пока работает функция bar().

Совершенно понятно, что bar() должна принимать параметр типа std::shared_ptr<A>; это обеспечит неприкосновенность A. Но откуда внутри A::foo() взять этот самый shared_ptr? Можно, конечно, добавить в A переменную-член типа std::shared_ptr<A>, но тогда A сам заблокирует свое уничтожение – он останется в памяти навсегда! Это совсем не то, что нам нужно!

Первое напрашивающееся решение – включить в A переменную-член типа std::weak_ptr<A>, указывающую на свой экземпляр A, и тогда вызов bar мог бы выглядеть так: bar(this->m_wptr.lock()). Однако это увеличивает синтаксические накладные расходы, а кроме того, не ясно, как инициализировать указатель m_wptr. C++ взял эту идею и встроил ее прямо в стандартную библиотеку!

```
template<class T>
class enable_shared_from_this {
  weak_ptr<T> m_weak;
public:
  enable_shared_from_this(const enable_shared_from_this&) {}
  enable_shared_from_this& operator=(const enable_shared_from_this&) {}
  shared_ptr<T> shared_from_this() const {
    return shared_ptr<T>(m_weak);
  }
};
```

Класс std::enable_shared_from_this<A> хранит нашу переменную-член типа std::weak_ptr<A> и предоставляет операцию «получить shared_ptr на самого себя» под именем x.shared_from_this(). В примере выше есть пара интересных деталей: во-первых, попытка вызвать x.shared_from_this() для объекта, который в данный момент не управляется системой shared_ptr, приведет к исключению std::bad_weak_ptr. Во-вторых, обратите внимание на пустой конструктор копирования и оператор присваивания копии. Пустые фигурные скобки в данном случае – это не то же самое, что =default! Если бы мы использовали =default, чтобы сделать копирование операциями по умолчанию, тогда копирование производилось бы почленно, и каждая операция копирования управляемого объекта создавала бы новый объект с копией поля m_weak оригинала; а это совсем не то, что нам нужно в данном случае. «Идентичность» части enable_shared_from_this объекта C++ неразрывно связана с его местоположением в памяти, и поэтому он не следует (и не должен следовать) правилам копирования и семантике значений, к которым мы обычно стремимся.

Последний вопрос, на который нужно ответить: как инициализировать член `m_weak` (который, как вы помните, является приватным членом; здесь имя `m_weak` используется исключительно для объяснений)? Дело в том, что конструктор `shared_ptr` включает несколько строк кода, проверяющих наследование `enable_shared_from_this<T>` типом `T` и устанавливающих его член `m_weak` через потайной ход. Обратите внимание, что наследование должно быть публичным и недвусмысленным, потому что с точки зрения правил C++ конструктор `shared_ptr` – это самая обычная пользовательская функция. Она не может проникнуть внутрь вашего класса, чтобы определить приватные базовые классы, или устранить неоднозначность между несколькими копиями `enable_shared_from_this`.

Эти ограничения предполагают, что наследование `enable_shared_from_this` должно быть публичным; и если ваш класс наследует `enable_shared_from_this`, наследовать его тоже необходимо публично; а чтобы избавиться от лишних сложностей, желательно наследовать `enable_shared_from_this` как можно ближе к листьям иерархии наследования. Если вы не будете создавать глубоких иерархий наследования, эти правила не покажутся вам сложными!

Теперь соединим все, что мы узнали о `enable_shared_from_this`, в один пример:

```
  struct Widget : std::enable_shared_from_this<Widget> {
    template<class F>
    void call_on_me(const F& f) {
      f(this->shared_from_this());
    }
  };

  void test() {
    auto sa = std::make_shared<Widget>();

    assert(sa.use_count() == 1);
    sa->call_on_me([](auto sb) {
      assert(sb.use_count() == 2);
    });

    Widget w;
    try {
      w.call_on_me([](auto) {});
    } catch (const std::bad_weak_ptr&) {
      puts("Caught!");
  }
}
```

Странно рекурсивный шаблон проектирования

Возможно ,вы заметили, особенно после предыдущего примера, что, наследуя `enable_shared_from_this`, вы должны указывать имя своего базового класса в списке параметров типов! Этот шаблон «`X` наследует `A<X>`» известен как «Странно рекурсивный шаблон проектирования» (Curiously Recurring Template Pattern, CRTP). Он используется, когда какой-то аспект базового класса зависит от производного класса. Например, в данном случае имя производного класса внедряется в тип возвращаемого значения метода `shared_from_this`.

Другой распространенный случай применения CRTP – когда требуется включить в базовый класс некоторое поведение производного класса. Например, используя CRTP, можно написать базовый шаблонный класс с оператором `operator+`, возвращающим значение любого производного класса, реализующего `operator+=` и конструктор копирования. Обратите внимание на обязательное приведение `static_cast` типа `addable<Derived>` к `Derived`, чтобы обеспечить вызов конструктора копирования класса `Derived`, а не базового класса `addable<Derived>`:

```
template<class Derived>
class addable {
public:
  auto operator+(const Derived& rhs) const {
    Derived lhs = static_cast<const Derived&>(*this);
    lhs += rhs;
    return lhs;
  }
};
```

Фактически это точная реализация службы `boost::addable` из библиотеки Boost Operators, за исключением того, что `boost::addable` использует так называемый «трюк Бартона-Накмена» (Barton-Nackman trick), чтобы сделать свой `operator+` дружественной свободной функцией, а не функцией-членом:

```
template<class Derived>
class addable {
public:
  friend auto operator+(Derived lhs, const Derived& rhs) {
    lhs += rhs;
    return lhs;
  }
};
```

Даже если вы нигде не используете `enable_shared_from_this` в своем коде, вы должны знать о «Странно рекурсивном шаблоне проектирования» и быть готовыми применить его на практике, когда потребуется «внедрить» некоторое поведение производного класса в метод базового класса.

Заключительное замечание

Мини-экосистема `shared_ptr`, `weak_ptr` и `enable_shared_from_this` – одна из лучших элементов современного C++; она придает коду безопасность языков с автоматизированной сборкой мусора, сохраняя скорость и предсказуемость моментов уничтожения объектов, которые всегда отличали C++. Однако старайтесь не злоупотреблять указателями `shared_ptr`! Большая часть кода на C++ не нуждается в применении `shared_ptr`, потому что не требует общего владения объектами, размещенными в куче. Вы в первую очередь должны избегать использования кучи (используя семантику значения); во вторую очередь вы должны гарантировать, что все объекты в куче имеют единственного владельца (с помощью `std::unique_ptr<T>`); и только когда ни то, ни другое невозможно, можно подумать о совместном владении и применении `std::shared_ptr<T>`.

Обозначение неисключительности с observer_ptr<T>

Итак, мы познакомились с двумя или тремя разными типами умных указателей (в зависимости от того, считать ли `weak_ptr` самостоятельным указателем или только лишь билетом на приобретение `shared_ptr`). Каждый из этих типов указателей несет в себе информацию, которую удобно использовать для управления жизненным циклом. Например, что можно сказать о семантике следующих двух функций C++, имея перед глазами только их сигнатуры?

```
void remusnoc(std::unique_ptr<Widget> p);

std::unique_ptr<Widget> recudorp();
```

Мы видим, что `remusnoc` принимает `unique_ptr` по значению, то есть функции `remusnoc` передается право владения управляемым объектом. Чтобы вызвать эту функцию, мы должны обладать *единоличным правом владения* объектом `Widget`, и после вызова функции мы уже не сможем получить доступ к нему. Мы не знаем, что сделает `remusnoc` с объектом `Widget` – уничтожит его, сохранит в памяти или передаст его в какой-то другой объект или поток выполнения; но что можно сказать определенно – объект *выходит из-под нашего контроля*. Функция `remusnoc` является потребителем объектов `Widget`.

Также можно сказать, что, вызывая `remusnoc`, мы должны единолично владеть объектом `Widget`, который был размещен в куче оператором `new` и может быть безопасно уничтожен оператором `delete`!

И наоборот: вызывая `recudorp`, я точно знаю, что любой `Widget`, который вернет эта функция, будет *принадлежать только мне*. Это не ссылка на чей-то чужой объект; это не указатель на некоторые статические данные. Этот объект размещен в куче и принадлежит только мне. Даже если я сразу же вызову метод

`.release()` полученного объекта и присвою простой указатель на некоторую «древнюю» структуру, я буду уверен, что это безопасно, потому что я – *единственный владелец* возвращаемого значения.

А что можно сказать о семантике такой функции?

```
void suougibma(Widget *p);
```

Она неоднозначна. Эта функция может принимать или не принимать на себя владение переданным указателем. Сказать что-то определенное можно, только исходя из описания `suougibma` или стилистических соглашений, принятых в нашем проекте (таких как «передача простого указателя никогда не означает передачу права владения», что вполне разумно), но, имея только сигнатуру, нельзя делать никаких предположений. Иначе говоря, `unique_ptr<T>` является *словарным типом* для выражения передачи владения, тогда как `T*` вообще не является словарным типом; в C++ это является эквивалентом бессмысленного слова или пятен Роршаха в том смысле, что два человека не обязательно увидят в нем один и тот же смысл.

Поэтому, обнаружив в своем коде множество невладеющих указателей, у вас может возникнуть желание иметь словарный тип, представляющий идею невладеющего указателя. (Конечно, первым вашим действием в такой ситуации должна стать организация передачи ссылок вместо указателей везде, где только возможно, но предположим, что вы уже исчерпали эту возможность.) Такой словарный тип существует, но он (пока) не включен в стандартную библиотеку C++ – как выразился Уолтер Браун (Walter Brown): «Это самый тупой из умных указателей», – и это всего лишь класс-обертка вокруг простого, невладеющего указателя:

```
template<typename T>
class observer_ptr {
  T *m_ptr = nullptr;
public:
  constexpr observer_ptr() noexcept = default;
  constexpr observer_ptr(T *p) noexcept : m_ptr(p) {}
  T *get() const noexcept { return m_ptr; }
  operator bool() const noexcept { return bool(get()); }
  T& operator*() const noexcept { return *get(); }
  T* operator->() const noexcept { return get(); }
};

void revresbo(observer_ptr<Widget> p);
```

Благодаря наличию `observer_ptr` в нашем арсенале можно с полной уверенностью сказать, что функция `revresbo` просто *наблюдает* (observes) за своим аргументом – она определенно не получает никаких прав на владение им. Фактически можно даже сказать, что она не хранит копию переданного указателя, потому что допустимость этого указателя зависит от жизненного цикла управляемого объекта, а `revresbo` явно заявляет, что никак не участвует в этом

цикле. Если бы она захотела как-то повлиять на жизненный цикл управляемого объекта, она заявила бы об этом явно, потребовав у вызывающего кода `unique_ptr` или `shared_ptr`. Потребовав `observer_ptr`, функция `revresbo` «отказывается» от любых притязаний на владение объектом.

Как я уже сказал, `observer_ptr` не является частью стандарта C++17. Одно из основных возражений, препятствующих этому, – его ужасное имя (поскольку оно никак не связано с шаблоном проектирования «Наблюдатель» (Observer)). Существует также большое количество знающих людей, которые могли бы сказать, что `T*` должен стать словарным типом для обозначения «невладеющего указателя» и что весь старый код, использующий `T*` для передачи прав владения, должен быть переписан или, по крайней мере, снабжен аннотациями в виде таких конструкций, как `owner<T*>`. В настоящее время именно этот подход рекомендуется редакторами «C++ Core Guidelines», включая создателя C++ Бьерна Страуструпа (Bjarne Stroustrup). Но, как бы то ни было, одно можно сказать наверняка: *никогда не используйте простые указатели для передачи права владения*!

Итоги

В этой главе мы познакомились с некоторыми аспектами умных указателей.

`std::unique_ptr<T>` – словарный тип для представления владеющего указателя и передачи права владения; старайтесь использовать его вместо `T*`. Подумайте о возможности использования `observer_ptr` в ситуациях, когда право владения не передается или когда применение простого указателя `T*` может выглядеть двусмысленно для читателя.

`std::shared_ptr<T>` – хороший (и стандартный) инструмент для случаев совместного владения, когда участие в жизненном цикле единственного управляемого объекта принимает множество заинтересованных сторон.

`std::weak_ptr<T>` – это «билет на `shared_ptr`»; он предоставляет метод `.lock()` вместо `operator*`. Если вашему классу потребуется получить указатель `shared_ptr` на себя самого, унаследуйте `std::enable_shared_from_this<T>`. Наследование должно быть публичным, и желательно как можно ближе к листьям в дереве наследования. Не злоупотребляйте этой возможностью в ситуациях, где явно не требуется совместное владение!

Никогда не пачкайте руки о простые указатели: используйте `make_unique` и `make_shared` для создания объектов в куче и управления ими. И вспоминайте о странно рекурсивном шаблоне проектирования, когда вам потребуется «внедрить» некоторое поведение производного класса в метод базового класса.

В следующей главе мы поговорим о другой разновидности «совместного использования», которая возникает в многопоточном программировании.

Глава 7

Конкуренция

В предыдущей главе мы узнали, как `std::shared_ptr<T>` реализует подсчет ссылок, чтобы дать возможность совместно управлять жизненным циклом объектов сразу нескольким заинтересованным сторонам, которые, возможно, даже не подозревают о существовании друг друга – например, они могут выполняться в разных потоках. В версиях C++ до C++11 это сразу же стало бы камнем преткновения: если один код уменьшает счетчик ссылок, а другой, выполняющийся в другом потоке, в это же время находится на полпути к уменьшению, разве не возникает состояния гонки и, как результат, небезопасного поведения?

В C++ до C++11 можно сразу ответить «да». (Фактически в C++ до C++11 отсутствовало стандартное понятие «потоков», поэтому на вопрос выше вполне можно было бы ответить, что он не имеет смысла.) Однако с выходом стандарта C++11 мы получили стандартную модель памяти, которая принимает во внимание такие понятия, как «потоки» и «безопасность в многопоточной среде», поэтому вопрос приобретает осмысленность и ответ на него: категоричное «нет»! Счетчик ссылок в `std::shared_ptr` гарантированно защищен от состояния гонки; и в этой главе вы увидите, как можно реализовать аналогичные потокобезопасные конструкции с использованием инструментов из стандартной библиотеки.

Эта глава охватывает следующие темы:

- различия между `volatile T` и `std::atomic<T>`;
- `std::mutex`, `std::lock_guard<M>` и `std::unique_lock<M>`;
- `std::recursive_mutex` и `std::shared_mutex`;
- `std::condition_variable` и `std::condition_variable_any`;
- `std::promise<T>` и `std::future<T>`;
- `std::thread` и `std::async`;
- опасности применения типа `std::async` и как сконструировать пул потоков для его замены.

Проблемы с volatile

Если последние десять лет вы провели в анабиозе или программировали на старом добром C, у вас может возникнуть вопрос: «Какие проблемы могут быть

связаны с ключевым словом volatile? Когда я хочу гарантировать сохранение значения в памяти, я использую volatile».

Официальная семантика volatile заключается в том, что доступ к памяти происходит в строгом соответствии с правилами абстрактной машины, то есть компилятору запрещено переупорядочивать обращения к таким переменным или объединять несколько обращений в одно. Например, компилятор не должен предполагать, что значение x останется неизменным между двумя операциями чтения; он обязан генерировать машинный код, выполняющий два чтения из памяти, как в примере ниже:

```
volatile int& x = memory_mapped_register_x();
volatile bool& y = memory_mapped_register_y();
int stack;

stack = x;  // чтение
y = true;   // запись
stack += x; // чтение
```

Если бы переменная x не была объявлена изменчивой (volatile), тогда компилятор мог бы переупорядочить инструкции, как показано ниже:

```
stack = 2*x; // чтение
y = true;    // запись
```

Компилятор мог бы поступить так (если бы x не была объявлена изменчивой), потому что запись в логическую переменную y никак не влияет на содержимое x. Но, так как x объявлена изменчивой, подобная оптимизация переупорядочением запрещена.

Чаще всего ключевое слово volatile используется по причине, которая вытекает из имен, использованных в примере выше: когда вы работаете напрямую с аппаратурой, такой как регистры, отображаемые в память, когда что-то на уровне исходного кода выглядит как операции чтения и записи, но в действительности отображается в более сложные операции с аппаратурой. В предыдущем примере переменная x может служить представлением одного из аппаратных буферов, а запись в y может служить сигналом аппаратуре загрузить следующие четыре байта данных в регистр x. Это можно рассматривать как перегрузку оператора, но на аппаратном уровне. А если фраза «перегрузка оператора на аппаратном уровне» кажется вам сумасшедшей, то, вероятно, у вас нет причин использовать volatile в своих программах!

Вот что делает ключевое слово volatile. Но почему его нельзя использовать, чтобы сделать наши программы безопасными в многопоточном окружении? По сути, проблема volatile в том, то это очень старый инструмент. Это ключевое слово существовало еще до того, как C++ отделился от C, и присутствовало еще в C, введенное стандартом 1989 года. В то время мало кого волновали вопросы многопоточности и компиляторы были намного проще, в том смысле, что некоторые потенциально проблематичные оптимизации

еще не были придуманы. К концу 1990-х – началу 2000-х, когда отсутствие в C++ модели памяти с поддержкой многопоточности превратилось в серьезную проблему, стало уже слишком поздно переделывать `volatile` для поддержки потокобезопасного доступа к памяти, потому что все производители компиляторов уже реализовали ключевое слово `volatile` и документально закрепили его действие. Смена правил в тот момент нарушила бы работоспособность огромного объема кода, причем эта беда затронула бы в основном код низкоуровневого аппаратного интерфейса, что было бы крайне нежелательно.

Вот несколько примеров, демонстрирующих гарантии, которые хотелось бы иметь для безопасного доступа к памяти в многопоточном окружении:

```
// Глобальные переменные:
int64_t x = 0;
bool y = false;

void thread_A() {
  x = 0x42'00000042;
  y = true;
}

void thread_B() {
  if (x) {
    assert(x == 0x42'00000042);
  }
}

void thread_C() {
  if (y) {
    assert(x == 0x42'00000042);
  }
}
```

Допустим, что `thread_A`, `thread_B` и `thread_C` были вызваны одновременно из разных потоков выполнения. Какие ошибки мог бы породить этот код? Во-первых, проверка в `thread_B` обнаружит в `x` ноль или `or 0x42'00000042`. Однако в 32-разрядном компьютере ситуация сложнее; компилятор может реализовать операцию присваивания в `thread_A` в виде пары инструкций записи, записав сначала число 0x42 в старшую половину `x`; а затем число 0x42 – в младшую половину `x`. Если проверка в `thread_B` запустится в нужный (для ошибки) момент, она может обнаружить в `x` значение `0x42'00000000`. Объявление переменной `x` изменчиво здесь не поможет; фактически это вообще ничего не даст, потому что 32-разрядная аппаратура просто не поддерживает эту операцию! Было бы хорошо, если бы компилятор обнаруживал попытку использовать атомарное 64-разрядное присваивание и выводил ошибку компиляции, сообщая о невозможности воплощения наших намерений. Иными словами, доступ к переменной, объявленной как `volatile`, необязательно будет *атомарным*. На практике они часто являются атомарными и поэтому не требуют

использования volatile, но это поведение не является гарантированным, и иногда приходится опускаться до уровня машинного кода, чтобы выяснить, действительно ли получился ожидаемый код. Нам хотелось бы иметь гарантии, что операции будут выполняться атомарно, а в случае невозможности этого получать ошибки компиляции.

Теперь рассмотрим функцию thread_C. Она сначала проверяет значение y, и если оно равно true, в переменную x уже должно быть записано окончательное числовое значение. То есть функция проверяет, что запись в x «произошла до» записи в y. Это верно с точки зрения thread_A, по крайней мере, если обе переменные, x и y, объявлены изменчивыми (volatile), потому что, как мы видели, в этом случае компилятору запрещается переупорядочивать операции доступа. Однако это необязательно будет верно с точки зрения thread_C! Если физически thread_C выполняется на другом процессоре, с собственным кешем данных, тогда он может узнать об обновлении значений x и y в разные моменты времени, в зависимости от особенностей обновления соответствующих блоков кеша. Хотелось бы иметь возможность потребовать от компилятора, чтобы тот гарантировал загрузку всего кеша вместе с загрузкой y и мы никогда не получали бы «устаревшего» значения x. Однако на некоторых процессорных архитектурах для этого требуется добавлять в код специальные инструкции или дополнительную логику работы с барьерами в памяти. Компиляторы *не генерируют* такие инструкции для «старорежимного» ключевого слова volatile, потому что многопоточности уделялось мало внимания, когда оно появилось; а кроме того, такие инструкции могут замедлять операции доступа к volatile-переменным и даже нарушать работоспособность существующего низкоуровневого кода, где volatile используется в старом его понимании. То есть проблема никуда не исчезла, потому что даже при соблюдении последовательности операций с переменными, объявленными с ключевым словом volatile, в одном потоке их обновленные значения могут появляться в другом потоке в ином порядке. Иначе говоря, volatile не гарантирует *последовательную согласованность*. Нам хотелось бы, чтобы доступ был последовательно согласован с другими операциями доступа.

Решение обеих проблем появилось в C++ в 2011 году. Это решение: std::atomic.

Использование std::atomic<T> для безопасного доступа в многопоточной среде

Начиная с C++11 заголовок <atomic> содержит определение шаблонного класса std::atomic<T>. Класс std::atomic можно рассматривать по-разному: как шаблонный класс, такой же как std::vector, с перегруженными операторами, гарантирующими безопасность в многопоточной среде; или как волшебное семейство встроенных типов, имена которых просто содержат угловые скобки. Последний способ более оправдан, потому что предполагает, и небезос-

новательно, что `std::atomic` частично встроен в компилятор и, как правило, компилятор генерирует наиболее оптимальный код для атомарных операций. Последний способ также объясняет, чем `atomic` отличается от `vector`: в `std::vector<T>` шаблон `T` может быть любым типом. В `std::atomic<T>` шаблон `T` тоже может быть любым типом, но на практике желательно ограничиваться только узким кругом типов, для которых атомарность поддерживается наиболее просто. Подробнее об этом мы поговорим чуть ниже.

К узкому кругу типов относятся все целочисленные типы (по крайней мере, с размерностью не больше размера регистров процессора) и указатели. На основных платформах операции с объектами `std::atomic` этих типов будут выполняться в точности, как нам хотелось бы:

```
// Глобальные переменные:
std::atomic<int64_t> x = 0;
std::atomic<bool> y = false;

void thread_A() {
  x = 0x42'00000042; // атомарная операция!
  y = true; // атомарная операция!
}

void thread_B() {
  if (x) {
    // Присваивание переменной x выполняется атомарно.
    assert(x == 0x42'00000042);
  }
}

void thread_C() {
  if (y) {
    // Присваивание переменной x выполняется "до"
    // присваивания переменной y, даже с точки зрения
    // другого потока выполнения.
    assert(x == 0x42'00000042);
  }
}
```

`std::atomic<T>` перегружает оператор присваивания, реализуя атомарную и потокобезопасную версию; а также операторы `++`, `--`, `+=` и `-=`; и для целочисленных типов, также операторы `&=`, `|=` и `^=`.

Важно понимать разницу между объектами типа `std::atomic<T>` (которые концептуально находятся «там», в памяти) и короткоживущими значениями типа `T` (находящимися «прямо здесь», под рукой; например, в регистрах процессора). Так, например, для `std::atomic<int>` не существует оператора присваивания копированием:

```
std::atomic<int> a, b;
a = b; // НЕ КОМПИЛИРУЕТСЯ!
```

Оператора присваивания копированием (как и оператора присваивания перемещением) не существует, потому что эта операция не имеет однозначного толкования: имел ли в виду программист, что компьютер должен загрузить значение `b` в регистр и затем сохранить значение этого регистра в `a`? В данном случае производятся две атомарные операции, а не одна! Или, может быть, программист имел в виду, что компьютер должен скопировать значение из `b` в `a` в рамках одной атомарной операции? Но в этом случае происходит обращение к двум разным участкам в памяти, и большинство компьютеров не поддерживает такого атомарного копирования. Поэтому C++ требует явно описать, что вы имеете в виду: одно атомарное чтение из объекта `b` в регистр (в C++ представляется неатомарной переменной на стеке) с последующей атомарной записью в объект `a`:

```
int shortlived = b; // атомарное чтение
a = shortlived;     // атомарная запись
```

`std::atomic<T>` поддерживает функции-члены `.load()` и `.store(v)` для тех программистов, кто хочет точно видеть, что делается в каждом шаге. Пользоваться ими совсем необязательно:

```
int shortlived = b.load(); // атомарное чтение
a.store(shortlived);       // атомарная запись
```

Фактически, используя эти функции-члены, присваивание *можно* записать в одной строке кода, как `b.store(a.load());`, но я не советую поступать так. Запись вызовов двух функций в одной строке не означает, что они непременно выполнятся «друг за другом», и *определенно* не означает, что вызовы произойдут «атомарно» (как говорилось выше, большая часть аппаратного обеспечения компьютеров не поддерживает такой возможности). И такая форма записи может *вводить в заблуждение*, заставляя думать, что вызовы происходят «вместе».

Работа с многопоточным кодом и без того сложна, поэтому старайтесь в одной строке записывать только одну операцию. Программирование нескольких операций в одной строке увеличивает вероятность появления ошибок. Записывайте по одной атомарной операции в каждой строке; постепенно вы поймете, что такой подход помогает упорядочить процесс мышления и упрощает чтение кода.

Атомарное выполнение сложных операций

Возможно, вы заметили, что в списке перегруженных операторов, приводившемся в предыдущем разделе, отсутствуют операторы `*=`, `/=`, `%=`, `<<=` и `>>=`. Эти операторы не поддерживаются типом `std::atomic<int>` и всеми остальными целочисленными атомарными типами, потому что на существующем аппаратном обеспечении трудно обеспечить их эффективную реализацию. Однако даже операции, включенные в `std::atomic<int>`, нередко требуют применения приемов, удорожающих реализацию.

Допустим, что наш процессор не поддерживает «атомарной инструкции умножения», но нам крайне необходимо реализовать operator*=. Как это сделать? Вся хитрость заключается в использовании простой атомарной операции, известной как «сравнить и поменять местами».

```
std::atomic<int> a = 6;

a *= 9; // Это недопустимо.

// Зато можно так:
int expected, desired;
do {
  expected = a.load();
  desired = expected * 9;
} while (!a.compare_exchange_weak(expected, desired));

// В конце этого цикла значение a будет
// "атомарно" умножено на 9.
```

Метод a.compare_exchange_weak(expected, desired) получает значение a и, если оно равно expected, записывает в a новое значение desired; иначе ничего не делается. Значение a возвращает true, если значение desired было записано, и false в противном случае.

Но здесь происходит еще кое-что. Обратите внимание, что в каждой итерации предыдущего цикла выполняется загрузка a в переменную expected, и функция compare_exchange_weak тоже загружает его, чтобы сравнить с expected. Начиная со второй итерации было бы лучше не загружать a повторно, а просто записать в expected значение, которое прочитала функция compare_exchange_weak. К счастью, создатели a.compare_exchange_weak(expected, desired) предвосхитили наше желание и реализовали запись в expected загруженного значения, если функция должна вернуть false. То есть в сочетании с compare_exchange_weak мы должны использовать значение expected, допускающее возможность изменения, потому что функция принимает его по ссылке.

Поэтому предыдущий пример можно переписать так:

```
int expected = a.load();
while (!a.compare_exchange_weak(expected, expected * 9)) {
  // продолжение цикла
}
```

Переменная desired фактически не нужна, в предыдущем примере она использовалась только лишь потому, что дополнительная инструкция if делает код понятнее.

Тип std::atomic имеет маленький секрет: большинство составных операций присваивания реализовано с применением цикла сравнения-замены, как в примере выше. На процессорах с архитектурой RISC эта реализация используется почти всегда. На процессорах x86 такая реализация используется,

только если вы хотите использовать значение, возвращаемое оператором, как в инструкции `x = (a += b)`.

Если атомарная переменная нечасто изменяется другими потоками, наличие цикла сравнения-замены почти не сказывается на производительности. Но если `a` изменяется часто, тогда может потребоваться несколько итераций, прежде чем цикл преуспеет. В абсолютно патологических ситуациях можно даже наблюдать зацикливание потока; он может продолжать выполнять цикл снова и снова, пока не исчезнут конфликты с другими потоками. Но обратите внимание, что `compare_exchange_weak` возвращает `false`, только когда значение `a` в памяти изменилось; а это значит, что другие потоки выполнения продвинулись немного вперед. Циклы сравнения-замены сами по себе никогда не заставят программу войти в состояние, когда никакой поток не сможет продвинуться вперед (состояние, известное как «динамическая взаимоблокировка» – «livelock»).

Предыдущий абзац звучит страшнее, чем есть на самом деле. В целом не стоит беспокоиться о патологическом поведении, потому что оно проявляется только при очень высокой конкуренции, но даже тогда не вызывает сколько-нибудь серьезных проблем. Главное, что вы должны вынести из этого раздела, – как можно использовать цикл сравнения-замены для реализации сложных, не встроенных «атомарных» операций в объектах `atomic<T>`. Просто запомните порядок следования параметров в `a.compare_exchange_weak(expected, desired)` и что она делает с `a`: «если `a` имеет значение `expected`, записать в `a` значение `desired`».

Большие атомарные типы

Компилятор распознает и генерирует оптимальный код для `std::atomic<T>`, когда `T` является целочисленным типом (включая `bool`) или типом указателя, таким как `void *`. А если `T` – большой тип, такой как `int[100]`? В таких случаях компилятор обычно вызывает подпрограммы в библиотеке времени выполнения C++, которые производят присваивание под защитой мьютекса. (Мы рассмотрим мьютексы чуть ниже.) Поскольку присваивание выполняется в библиотеке, которая не знает, как копировать произвольные пользовательские типы, стандарт C++17 допускает применение `std::atomic<T>` только к типам, которые поддерживают тривиальное копирование, то есть безопасно могут копироваться с помощью `memcpy`. Поэтому, если вам понадобится тип `std::atomic<std::string>`, считайте, что вам не повезло – вы должны будете написать его сами.

Другая проблема, с которой можно столкнуться при использовании больших (тривиально копируемых) типов с `std::atomic`, состоит в том, что соответствующие процедуры в библиотеке времени выполнения C++ находятся в другом месте, отдельно от остальной части стандартной библиотеки C++. На некоторых платформах вам придется добавить `-latomic` в командную строку компоновщика. Но эта проблема проявляется только при использовании боль-

ших типов с `std::atomic`, а если этого не делать, тогда и проблема отпадает сама собой.

Теперь посмотрим, как написать атомарный класс строк!

Поочередное выполнение с std::mutex

Допустим, нам нужно написать класс, своим поведением напоминающий тип `std::atomic<std::string>`, если бы таковой существовал. То есть нам нужен класс, поддерживающий атомарные операции чтения и записи, безопасные в многопоточной среде, чтобы при одновременном обращении двух потоков к `std::string` ни один из них не мог бы увидеть строку, пока другой не закончит запись в память, как это имело место в примере с типом `int64_t`, в разделе «Проблемы с volatile» выше.

Лучше всего для реализации такого класса использовать тип `std::mutex` из стандартной библиотеки. Имя «mutex» (мьютекс) настолько широко используется в технических кругах, что уже воспринимается как самостоятельное слово, но первоначально это имя произошло от двух слов «mutual exclusion» (взаимоисключающий). Это связано с тем, что мьютекс позволяет гарантировать исключительный доступ к некоторому сегменту кода для одного потока – то есть гарантировать невозможность ситуации, когда «потоки A и B одновременно выполняют этот код».

В начале такого критического сегмента кода, чтобы показать нежелательность пересечения с любым другим потоком, мы *приобретаем блокировку* на соответствующем мьютексе. Покидая этот сегмент кода, мы *освобождаем блокировку*. Библиотека сама позаботится о том, чтобы никакие два потока не смогли в одно и то же время владеть блокировкой для одного и того же мьютекса. В частности, это означает, что если поток B подошел к точке приобретения блокировки, когда поток A уже захватил ее, поток B должен будет *дождаться*, пока поток A завершит выполнение критического сегмента кода и освободит блокировку. Пока поток A удерживает блокировку, работа потока B *блокируется*; поэтому такое поведение часто называют *блокирующим*.

Фразу «приобрести блокировку на мьютексе» часто сокращают до «запереть мьютекс», а «освободить блокировку» – до «отпереть мьютекс».

Иногда (хотя и редко) бывает желательно проверить текущее состояние мьютекса. Для этой цели `std::mutex` поддерживает не только функции-члены `.lock()` и `.unlock()`, но также функцию-член `.try_lock()`, которая возвращает `true`, если мьютекс удалось запереть, и `false`, если мьютекс в этот момент уже заперт каким-то другим потоком.

В некоторых языках программирования, таких как Java, каждый объект снабжается своим мьютексом; с его помощью, например, в Java реализуются блоки `synchronized`. В C++ мьютекс является отдельным типом объектов; чтобы воспользоваться мьютексом для управления сегментом кода, вы должны создать отдельный объект мьютекса со своим жизненным циклом. Куда следует поместить мьютекс, чтобы он был единственным объектом и доступным всем,

кто пользуется им? Иногда, если существует только один критический сегмент кода, нуждающийся в защите, мьютекс можно заключить в область видимости функции в виде статической переменной:


```
void log(const char *message)
{
  static std::mutex m;
  m.lock(); // чтобы исключить перемешивания сообщений в stdout
  puts(message);
  m.unlock();
}
```

Ключевое слово `static` здесь очень важно! Если опустить его, `m` превратится в обычную локальную переменную на стеке, и каждый поток, вызывающий `log`, получит свою отдельную копию `m`. Это никак не помогло бы нам в достижении нашей цели, потому что библиотека гарантирует невозможность захвата двумя потоками *одного и того же* мьютекса. Но если каждый поток будет запирать и отпирать свой собственный мьютекс, библиотека не сможет ничего противопоставить этому; потому что нет борьбы за обладание мьютексами.

Чтобы обеспечить взаимоисключающее выполнение для двух разных функций, то есть чтобы только один поток мог войти в функцию `log1` или `log2`, объект мьютекса следует разместить в области видимости, где он будет доступен обеим функциям:

```
static std::mutex m;

void log1(const char *message) {
  m.lock();
  printf("LOG1: %s\n", message);
  m.unlock();
}

void log2(const char *message) {
  m.lock();
  printf("LOG2: %s\n", message);
  m.unlock();
}
```

Вообще говоря, когда возникает подобная необходимость, предпочтительнее устранить глобальную переменную, создав класс и включив в него объект мьютекса как переменную-член, например:

```
struct Logger {
  std::mutex m_mtx;

  void log1(const char *message) {
    m_mtx.lock();
    printf("LOG1: %s\n", message);
    m_mtx.unlock();
```

```
  }

  void log2(const char *message) {
    m_mtx.lock();
    printf("LOG2: %s\n", message);
    m_mtx.unlock();
  }
};
```

Теперь сообщения, выводимые с помощью разных экземпляров `Logger`, могут перемежаться, но при конкурентном доступе одному и тому же экземпляру `Logger` будет запираться один и тот же мьютекс `m_mtx`, поэтому разные потоки будут блокировать друг друга и по очереди выполнять критические функции `log1` и `log2`.

Правильный порядок «приобретения блокировок»

Как рассказывалось в главе 6 «Умные указатели», одной из главных проблем программ, написанных на C и на C++ «старого стиля», является наличие ошибок, связанных с указателями, таких как утечки памяти, повторное освобождение и повреждение кучи, и одним из способов их устранения из программ на C++ «нового стиля» является использование типов RAII, таких как `std::unique_ptr<T>`. Многопоточное программирование с применением простых мьютексов имеет схожие проблемы с распределением динамической памяти в куче с использованием простых указателей.

- **Утечка блокировок:** можно приобрести блокировку для какого-то определенного мьютекса и по ошибке забыть написать код, освобождающий ее.
- **Утечка блокировок:** можно написать код, освобождающий блокировку, но из-за исключения или преждевременного возврата он не будет выполнен, и мьютекс останется запертым!
- **Использование защищенных переменных без приобретения блокировки:** так как обычный мьютекс – это всего лишь переменная, она физически не связана с переменными, «находящимися под защитой» этого мьютекса. Можно по ошибке обратиться к одной из таких переменных, не приобретя блокировку.
- **Взаимоблокировка:** представьте, что поток A запер мьютекс 1, а поток B запер мьютекс 2. Затем поток A попытался запереть мьютекс 2 (и заблокировался); и пока поток A оставался заблокированным, поток B попытался запереть мьютекс 1 (и тоже заблокировался). Теперь оба потока оказались заблокированными навечно.

Это далеко не исчерпывающий список ловушек многопоточного выполнения; например, мы уже коротко обсудили состояние «динамической взаимоблокировки» в сочетании с типом `std::atomic<T>`. Более полное перечисление

ловушек многопоточного программирования и как их избежать, вы найдете в специализированных книгах, посвященных многопоточному, или конкурентному, программированию.

В стандартной библиотеке C++ имеется несколько инструментов, которые могут помочь нам избавить многопоточные программы от этих проблем. В отличие от управления памятью, стандартные решения перечисленных проблем не дают 100% гарантии – многопоточное программирование намного сложнее однопоточного, и в отношении него действует одно хорошее правило: не прибегать к нему, если многопоточность не является абсолютно необходимой. Но если без многопоточности не обойтись, стандартная библиотека может оказать вам некоторую помощь.

Ошибки, имеющие отношение к «утечкам блокировок», можно устранить использованием идеи RAII. Возможно, вы обратили внимание, что я постоянно использую фразу «приобрести блокировку» вместо «заблокировать»; теперь я объясню – почему. Слово «заблокировать» – это глагол, точно соответствующий коду на C++ `mtx.lock()`. Но в словосочетании «приобрести блокировку» слово «блокировка» – это существительное. Давайте придумаем тип, воплощающий идею существительного «блокировка»; то есть превращающий ее в существительное (класс RAII), а не глагол (метод класса не-RAII):

```
template<typename M>
class unique_lock {
  M *m_mtx = nullptr;
  bool m_locked = false;
public:
  constexpr unique_lock() noexcept = default;
  constexpr unique_lock(M *p) noexcept : m_mtx(p) {}

  M *mutex() const noexcept { return m_mtx; }
  bool owns_lock() const noexcept { return m_locked; }

  void lock() { m_mtx->lock(); m_locked = true; }
  void unlock() { m_mtx->unlock(); m_locked = false; }

  unique_lock(unique_lock&& rhs) noexcept {
    m_mtx = std::exchange(rhs.m_mtx, nullptr);
    m_locked = std::exchange(rhs.m_locked, false);
  }

  unique_lock& operator=(unique_lock&& rhs) {
    if (m_locked) {
      unlock();
    }
    m_mtx = std::exchange(rhs.m_mtx, nullptr);
    m_locked = std::exchange(rhs.m_locked, false);
    return *this;
  }

  ~unique_lock() {
```

```
    if (m_locked) {
      unlock();
    }
  }
};
```

Как следует из имени, `std::unique_lock<M>` – это RAII-класс, описывающий «уникальное владение» в духе `std::unique_ptr<T>`. Если придерживаться существительного `unique_ptr` вместо глаголов `new` и `delete`, вы никогда не забудете освободить указатель; а если придерживаться существительного `unique_lock` вместо глаголов `lock` и `unlock`, вы никогда не забудете освободить блокировку.

`std::unique_lock<M>` реализует функции-члены `.lock()` и `.unlock()`, но вам редко придется их использовать. Они могут пригодиться, если потребуется приобрести или освободить блокировку в середине блока кода, далеко от точки уничтожения объекта `unique_lock`. В следующем разделе мы увидим также функцию, которая принимает заблокированный экземпляр `unique_lock`, освобождает блокировку и вновь приобретает ее в процессе своей работы.

Обратите внимание, что `unique_lock`, как поддерживающий возможность перемещения, должен иметь «пустое» состояние, в точности как `unique_ptr`. В большинстве случаев не требуется перемещать блокировки; вы просто будете безусловно приобретать блокировку в начале некоторой области и затем так же безусловно освобождать ее в конце. В такой ситуации `std::lock_guard<M>`. `lock_guard` сильно напоминает `unique_lock`, но не поддерживает перемещения и не имеет функций-членов `.lock()` и `.unlock()`. Поэтому ему не нужен член `m_locked`, и его деструктор может без лишних проверок отпереть мьютекс, защищающий объект.

В обоих случаях (`unique_lock` и `lock_guard`) шаблонный класс параметризуется типом мьютекса. (Чуть ниже мы познакомимся еще с парой типов мьютексов, но на практике почти всегда вы будете использовать `std::mutex`.) Стандарт C++17 добавил в язык новую особенность с названием *вывод аргумента шаблонного класса*, которая в большинстве случаев позволяет избавиться от параметра шаблона, и вместо `std::unique_lock<std::mutex>` просто писать `std::unique_lock`. Это один из немногих случаев, когда я лично рекомендовал бы положиться на механизм вывода аргумента шаблонного класса, потому что дополнительный параметр типа `std::mutex` почти не дает читателю полезной информации.

Рассмотрим несколько примеров `std::lock_guard` с использованием вывода аргумента шаблонного класса и без него:

```
struct Lockbox {
  std::mutex m_mtx;
  int m_value = 0;

  void locked_increment() {
```

```
    std::lock_guard<std::mutex> lk(m_mtx);
    m_value += 1;
  }

  void locked_decrement() {
    std::lock_guard lk(m_mtx); // Только в C++17
    m_value -= 1;
  }
};
```

Прежде чем перейти к похожим практическим примерам использования `std::unique_lock`, разберем некоторые причины в пользу `std::unique_lock`.

Всегда связывайте мьютекс с управляемыми данными

Взгляните на следующую черновую реализацию потокобезопасного класса `StreamingAverage`. В ней имеется ошибка. Сможете найти ее?

```
class StreamingAverage {
  double m_sum = 0;
  int m_count = 0;
  double m_last_average = 0;
  std::mutex m_mtx;
public:
  // Вызывается из единственного потока-производителя
  void add_value(double x) {
    std::lock_guard lk(m_mtx);
    m_sum += x;
    m_count += 1; // A
  }

  // Вызывается из единственного потока-потребителя
  double get_current_average() {
    std::lock_guard lk(m_mtx);
    m_last_average = m_sum / m_count; // B
    return m_last_average;
  }

  // Вызывается из единственного потока-потребителя
  double get_last_average() const {
    return m_last_average; // C
  }

  // Вызывается из единственного потока-потребителя
  double get_current_count() const {
    return m_count; // D
  }
};
```

Ошибка в строке с комментарием `// A`, которая записывает новое значение в `this->m_count` в потоке-производителе. Она подвержена состоянию гонки со строкой `// D`, которая читает `this->m_count` в потоке-потребителе. Строка `// A` правильно приобретет блокировку `this->m_mtx` перед записью, но строка `// D` отказывается приобретать блокировку, а значит, благополучно прочитает `m_count`, даже когда строка `// A` выполнила операцию записи только наполовину.

На первый взгляд кажется, что строки `// B` и `// C` содержат ту же ошибку. Но в строке `// C` блокировка не нужна; тогда почему она нужна в строке `// D`? Дело все в том, что строка `// C` вызывается только из потока-потребителя, того же потока, который выполняет запись в `m_last_average` в строке `// B`. Так как строки `// B` и `// C` выполняются в единственном потоке-потребителе, они не могут выполняться одновременно – по крайней мере, пока поведение программы соответствует комментариям! (Допустим, что комментарии в коде верны. К сожалению, на практике это не всегда так, но в данном примере предположим, что комментарии отражают истину.)

У нас есть повод для сомнений: для доступа к `m_sum` или `m_count` необходимо заблокировать `m_mtx`, но этого можно не делать перед обращением к `m_last_average`. При усложнении класса в нем может появиться еще несколько мьютексов (хотя в данный момент это явно выглядит как нарушение принципа единственной ответственности, и, вероятно, будет произведен рефакторинг с выделением меньших компонентов). Поэтому на практике желательно помещать мьютексы как можно ближе к переменным, которые он «защищает». Часто для этого достаточно с особым вниманием отнестись к выбору имен:

```
class StreamingAverage {
  double m_sum = 0;
  int m_count = 0;
  double m_last_average = 0;
  std::mutex m_sum_count_mtx;

  // ...
};
```

Еще лучше – использовать вложенную структуру:

```
class StreamingAverage {
  struct {
    double sum = 0;
    int count = 0;
    std::mutex mtx;
  } m_guarded_sc;
  double m_last_average = 0;

  // ...
};
```

Выше мы понадеялись, что если вынудить программиста писать `this->m_guarded_sc.sum`, он запомнит, что предварительно должен приобрести

блокировку `this->m_guarded_sc.mtx`. Чтобы избежать повторного ввода `m_guarded_sc` повсюду в коде, мы могли бы использовать GNU-расширение «анонимные члены структуры»; но это противоречило бы цели данного подхода – в каждой точке, где происходит доступ к данным, использовать слово «guarded» (защищенный), напоминающее программисту о необходимости приобретения блокировки `this->m_guarded_sc.mtx`.

Еще более надежный, но менее гибкий способ – поместить мьютекс в класс, который открывает доступ к своим приватным членам, только когда мьютекс заперт, возвращая дескриптор RAII. Класс, возвращающий такой дескриптор, мог бы выглядеть примерно так:

```
template<class Data>
class Guarded {
  std::mutex m_mtx;
  Data m_data;

  class Handle {
    std::unique_lock<std::mutex> m_lk;
    Data *m_ptr;
  public:
    Handle(std::unique_lock<std::mutex> lk, Data *p) :
      m_lk(std::move(lk)), m_ptr(p) {}
    auto operator->() const { return m_ptr; }
  };
public:
  Handle lock() {
    std::unique_lock lk(m_mtx);
    return Handle{std::move(lk), &m_data};
  }
};
```

А наш класс `StreamingAverage` мог бы использовать его так:

```
class StreamingAverage {
  struct Guts {
    double m_sum = 0;
    int m_count = 0;
  };
  Guarded<Guts> m_sc;
  double m_last_average = 0;

  // ...

  double get_current_average() {
    auto h = m_sc.lock();
    m_last_average = h->m_sum / h->m_count;
    return m_last_average;
  }
};
```

В предыдущем примере никакая функция-член класса `StreamingAverage` не сможет обратиться к `m_sum`, не заперев мьютекс `m_mtx`; доступ к защищаемому члену `m_sum` возможен только посредством RAII-типа `Handle`.

Этот шаблон проектирования включен в библиотеку Facebook Folly под именем `folly::Synchronized<T>`, и многие его варианты доступны в библиотеке libGuarded, разработчики которой – Ансел Сермерсхайм (Ansel Sermersheim) и Барбара Геллер (Barbara Geller).

Обратите внимание, как используется `std::unique_lock<std::mutex>` в классе `Handle`! Здесь мы использовали `unique_lock`, а не `lock_guard`, потому что нам необходимо иметь возможность передавать блокировку, возвращать ее из функций и т. д. – поэтому она должна быть перемещаемой. Это главная причина добавления `unique_lock` в наш арсенал.

Имейте в виду, что этот прием не решает всех проблем с блокировками – он помогает только в простейших случаях, когда программист просто «забывает запереть мьютекс», – и может стимулировать применение шаблонов программирования, которые влекут за собой ошибки конкурентного выполнения других видов. Например, взгляните на переделанный вариант `StreamingAverage::get_current_average`:

```
double get_sum() {
  return m_sc.lock()->m_sum;
}

int get_count() {
  return m_sc.lock()->m_count;
}

double get_current_average() {
  return get_sum() / get_count();
}
```

Поскольку здесь имеется два вызова `m_sc.lock()`, между записью в `m_sum` и чтением `m_count` появляется промежуток. Если в этом промежутке поток-производитель вызовет `add_value`, мы получим ошибочное среднее значение (уменьшенное в `m_count` раз). А если попытаемся «исправить» ошибку, добавив блокировку перед вычислениями, столкнемся с взаимоблокировкой:

```
double get_sum() {
  return m_sc.lock()->m_sum; // LOCK 2
}

int get_count() {
  return m_sc.lock()->m_count;
```

```
}

double get_current_average() {
  auto h = m_sc.lock(); // LOCK 1
  return get_sum() / get_count();
}
```

Строка с комментарием `// LOCK 1` заблокирует мьютекс; после этого строка с комментарием `// LOCK 2` попытается заблокировать мьютекс еще раз. Когда поток пытается запереть запертый мьютекс, его выполнение блокируется и возобновляется, только когда мьютекс будет отперт. То есть поток заблокируется и будет ждать, когда мьютекс освободится, чего никогда не произойдет, потому что блокировка удерживается этим же потоком!

Такие проблемы (взаимоблокировки с самим собой) должны, как правило, решаться внимательным отношением к программированию, то есть вы не должны пытаться приобрести блокировку, которую уже удерживаете! Но при таком подходе блокировки неизбежно становятся частью вашего дизайна. Избежать этой проблемы вам поможет стандартная библиотека в лице `recursive_mutex`.

Специальные типы мьютексов

Напомню, что `std::lock_guard<M>` и `std::unique_lock<M>` параметризуются типом мьютекса. До сих пор мы видели только тип `std::mutex`. Однако в стандартной библиотеке имеется несколько других типов мьютексов, которые могут пригодиться в определенных условиях.

`std::recursive_mutex` похож на `std::mutex`, но помнит, какой поток запер его. Если этот конкретный поток попытается приобрести блокировку повторно, рекурсивный мьютекс просто увеличит внутренний счетчик «количества приобретенных блокировок». Если какой-то другой поток попробует запереть рекурсивный мьютекс, он будет заблокирован, пока оригинальный поток не отопрет мьютекс соответствующее число раз.

`std::timed_mutex` похож на `std::mutex`, но поддерживает ограничение попытки приобрести блокировку по времени. Кроме обычной функции-члена `.try_lock()`, он имеет также `.try_lock_for()` и `.try_lock_until()`, которые используют стандартную библиотеку `<chrono>`. Вот пример использования `try_lock_for`:

```
std::timed_mutex m;
std::atomic<bool> ready = false;

std::thread thread_b([&]() {
  std::lock_guard lk(m);
  puts("Thread B got the lock.");
  ready = true;
  std::this_thread::sleep_for(100ms);
```

```
});

while (!ready) {
  puts("Thread A is waiting for thread B to launch.");
  std::this_thread::sleep_for(10ms);
}

while (!m.try_lock_for(10ms)) {
  puts("Thread A spent 10ms trying to get the lock and failed.");
}

puts("Thread A finally got the lock!");
m.unlock();
```

А вот пример использования `try_lock_until`:

```
std::timed_mutex m1, m2;
std::atomic<bool> ready = false;

std::thread thread_b([&]() {
  std::unique_lock lk1(m1);
  std::unique_lock lk2(m2);
  puts("Thread B got the locks.");
  ready = true;
  std::this_thread::sleep_for(50ms);
  lk1.unlock();
  std::this_thread::sleep_for(50ms);
});

while (!ready) {
  std::this_thread::sleep_for(10ms);
}


auto start_time = std::chrono::system_clock::now();
auto deadline = start_time + 100ms;

bool got_m1 = m1.try_lock_until(deadline);
auto elapsed_m1 = std::chrono::system_clock::now() - start_time;

bool got_m2 = m2.try_lock_until(deadline);
auto elapsed_m2 = std::chrono::system_clock::now() - start_time;

if (got_m1) {
  printf("Thread A got the first lock after %dms.\n",
  count_ms(elapsed_m1));
  m1.unlock();
}

if (got_m2) {
  printf("Thread A got the second lock after %dms.\n",
  count_ms(elapsed_m2));
```

```
  m2.unlock();
}
```

К слову сказать, используемая здесь функция `count_ms` – это всего лишь небольшое лямбда-выражение, которое постепенно вытесняет некоторые шаблоны, привычные для `<chrono>`:

```
auto count_ms = [](auto&& d) -> int {
  using namespace std::chrono;
  return duration_cast<milliseconds>(d).count();
};
```

Обратите внимание, что в обоих предыдущих примерах для синхронизации потоков `A` и `B` используется `std::atomic<bool>`. Мы просто инициализируем атомарную переменную значением `false` и затем выполняем цикл, пока она не получит значение `true`. В теле цикла выполняется вызов `std::this_thread::sleep_for`, чего достаточно, чтобы подсказать компилятору, что значение атомарной переменной может измениться. Будьте осторожны и никогда не пишите циклов опроса, которые не содержат вызова функции приостановки выполнения, потому что иначе компилятор может свернуть все последующие операции чтения в одну-единственную и бесконечный цикл.

`std::recursive_timed_mutex` сочетает в себе черты `recursive_mutex` и `timed_mutex`; он поддерживает семантику «счета» `recursive_mutex` плюс методы `try_lock_for` и `try_lock_until` от `timed_mutex`.

Тип `std::shared_mutex` имеет, возможно, недостаточно говорящее название. Он реализует поведение, которое в большинстве книг, посвященных конкурентному выполнению, называется *блокировкой для чтения/записи*. Определяющей характеристикой блокировки чтения/записи, или `shared_mutex`, является поддержка «блокирования» двумя разными способами. Можно приобрести обычную исключительную блокировку («для записи») вызовом `sm.lock()` или неисключительную («для чтения») блокировку вызовом `sm.lock_shared()`. Блокировку для чтения могут приобрести сразу несколько потоков; но если кто-то приобрел блокировку для чтения, никто не сможет приобрести ее для записи; а если кто-то приобрел ее для записи, никто не сможет приобрести ее ни для чтения, ни для записи. По сути, это те же правила , что определяют «состояние гонки» в модели памяти C++: два потока могут одновременно читать данные из одного объекта, при условии что в то же время нет потока, выполняющего запись. Тип `std::shared_mutex` добавляет в эту смесь безопасность. Он гарантирует, что если некоторый поток пытается выполнить запись (по крайней мере, если ведет себя добропорядочно и предварительно приобретает блокировку для записи на `std::shared_mutex`), он заблокируется до момента, когда все читающие потоки завершат чтение и операция записи не вызовет конфликтов.

`std::unique_lock<std::shared_mutex>` – это существительное, соответствующее исключительной («для записи») блокировке на `std::shared_mu-`

tex. Как нетрудно догадаться, стандартная библиотека поддерживает также std::shared_lock<std::shared_mutex> для воплощения идеи неисключительной («для чтения») блокировки в std::shared_mutex.

Повышение статуса блокировки для чтения/записи

Допустим, вы приобрели блокировку для чтения на shared_mutex (то есть для std::shared_lock<std::shared_mutex> lk вызвали lk.owns_lock()), и теперь вам нужно получить блокировку для записи. Можно ли «повысить статус» имеющейся блокировки?

Нет, нельзя. Представьте, что случится, если потоки A и B оба удерживают блокировку для чтения и одновременно попытаются повысить ее статус до блокировки для записи, не освобождая свои блокировки для чтения. Ни один из них не сможет приобрести блокировку для записи, и к тому же они попадут в состояние взаимоблокировки.

В действительности существуют сторонние библиотеки, пытающиеся решить эту проблему, как, например, boost::thread::upgrade_lock, которая работает с boost::thread::shared_mutex; но их обсуждение выходит далеко за рамки этой книги. Стандартное решение заключается в следующем: если вы удерживаете блокировку для чтения и хотите приобрести блокировку для записи, вы должны сначала освободить блокировку для чтения, а потом встать в очередь желающих получить блокировку для записи:

```
template<class M>
std::unique_lock<M> upgrade(std::shared_lock<M> lk)
{
  lk.unlock();
  // Здесь может вклиниться другой пишущий поток.
  return std::unique_lock<M>(*lk.mutex());
}
```

Понижение статуса блокировки для чтения/записи

Допустим, вы приобрели исключительную блокировку для чтения/записи на shared_mutex и теперь вам нужно получить блокировку для чтения. Можно ли «понизить статус» имеющейся блокировки?

В принципе, это возможно, как будто ничто не препятствует понижению статуса блокировки для записи до блокировки для чтения; но стандарт C++17 не предусматривает такой возможности непосредственно. Так же, как в случае с повышением статуса, можно использовать boost::thread::shared_mutex. Стандартное решение заключается в следующем: если вы удерживаете блокировку для записи и хотите приобрести блокировку для чтения, вы должны сначала освободить блокировку для записи, а потом встать в очередь желающих получить блокировку для чтения:

```
template<class M>
std::shared_lock<M> downgrade(std::unique_lock<M> lk)
```

```
{
  lk.unlock();
  // Здесь может вклиниться другой пишущий поток.
  return std::shared_lock<M>(*lk.mutex());
}
```

Как можно видеть на этих примерах, на данный момент `std::shared_mutex` из C++17 еще не готов ко всем перипетиям жизни. Если ваша архитектура вызовов требует такого переключения между блокировками чтения и записи, я настоятельно советую использовать что-нибудь вроде `boost::thread::shared_mutex`, который поставляется «с батарейками в комплекте».

Возможно, вы заметили, что новые читающие потоки могут продолжать приобретать блокировку для чтения, пока она удерживается каким-то конкретным потоком, но новые пишущие потоки лишены такой возможности. В результате этого пишущий поток может надолго заблокироваться из-за непрекращающейся последовательности читающих потоков, если реализация не предусматривает гарантий против замораживания пишущих потоков. `boost::thread::shared_mutex` дает такую гарантию (по крайней мере, если такая возможность предусматривается системным планировщиком). Стандартная формулировка `std::shared_mutex` не включает такой гарантии, даже притом, что любая реализация, допускающая такое подвешивание, должна считаться некачественной. Но на практике реализации стандартной библиотеки содержат `shared_mutex`, близкий по своим характеристикам к версии в библиотеке Boost, за исключением отсутствующих функций повышения/понижения статуса.

Ожидание условия

В разделе «Специальные типы мьютексов» мы запустили выполнение задания в отдельном потоке, а потом вынуждены были ждать, пока не закончится определенный этап инициализации. В том примере мы использовали цикл опроса на основе `std::atomic<bool>`. Но существует лучшее решение!

Проблема с нашим циклом опроса, приостанавливающимся на 50 миллисекунд, в том, что он всегда будет тратить на приостановку больше времени, чем нужно. Иногда наш поток будет возобновлять выполнение, обнаруживать, что условие его пробуждения не выполнено и снова приостанавливаться – это значит, что первая приостановка недостаточно долгая. Иногда поток будет возобновлять выполнение, обнаруживать, что условие было выполнено в течение последних 50 миллисекунд, но мы не можем сказать, когда именно, – это означает, что в среднем поток простаивает лишние 25 миллисекунд. Что бы ни случилось, вероятность возобновить выполнение *в нужный момент времени* ничтожно мала.

Чтобы не терять времени, нужен механизм, позволяющий отказаться от циклов опроса. В стандартной библиотеке есть такой механизм, позволяю-

щий прервать ожидание как раз в нужный момент времени; и называется он std::condition_variable.

Имея переменную cv типа std::condition_variable, наш поток мог бы приостановиться в «ожидании на» cv, вызвав метод cv.wait(lk). Вызов cv.notify_one() или cv.notify_all() возобновит выполнение одного или всех потоков, в данный момент ожидающих на cv. Но это не единственный способ возобновить выполнение потоков! Может так сложиться, что внешнее прерывание (например, сигнал POSIX) разбудит поток без вызова notify_one. Это явление называется *ложным пробуждением*. Обычно, чтобы защититься от ложных пробуждений, выполняется дополнительная проверка условия. Например, если поток ждет появления входных данных в буфере b, тогда после возобновления он должен проверить наличие данных вызовом b.empty() и, если буфер пуст, снова приостановиться.

По определению должен существовать какой-то другой поток, который поместит данные в b; то есть вызов b.empty() лучше выполнять под защитой некоторого мьютекса. То есть первое, что нужно сделать сразу после возобновления, – запереть этот мьютекс, и последнее, что нужно сделать непосредственно перед повторной приостановкой, – отпереть мьютекс. (Фактически вы должны автоматически отпирать мьютекс в ходе выполнения операции повторной простановки, чтобы никто не мог «проскользнуть», изменить b и вызвать cv.notify_one() до того, как поток приостановится.) Эта логическая цепочка объясняет, почему cv.wait(lk) принимает параметр lk, – это std::unique_lock<std::mutex>, который будет отпираться при приостановке и запираться в момент возобновления!

Вот пример ожидания некоторого условия. Сначала взгляните на реализацию с применением расточительного цикла опроса с переменной std::atomic:

```
std::atomic<bool> ready = false;

std::thread thread_b([&]() {
  prep_work();
  ready = true;
  main_work();
});

// Ждать готовности потока B.
while (!ready) {
  std::this_thread::sleep_for(10ms);
}
// Поток B завершил подготовку к работе.
```

А теперь на реализацию с использованием более эффективной условной переменой condition_variable:

```
bool ready = false; // не атомарная!
std::mutex ready_mutex;
```

```
std::condition_variable cv;

std::thread thread_b([&]() {
  prep_work();
  {
    std::lock_guard lk(ready_mutex);
    ready = true;
  }
  cv.notify_one();
  main_work();
});

// Ждать готовности потока B.
{
  std::unique_lock lk(ready_mutex);
  while (!ready) {
    cv.wait(lk);
  }
}
// Поток B завершил подготовку к работе.
```

Если ожидается, когда станет доступна для чтения структура, защищенная блокировкой для чтения/записи (то есть `std::shared_mutex`), тогда нужно передавать не `std::unique_lock<std::mutex>`, а `std::shared_lock<std::shared_mutex>`. Однако такое возможно, только (к сожалению, только) мы заранее планировали это и определили условную переменную с типом `std::condition_variable_any`, а не `std::condition_variable`. На практике вы едва ли заметите какую-либо разницу в производительности между типами `std::condition_variable_any` и `std::condition_variable`, а значит, старайтесь выбирать между ними, исходя из своих потребностей, или, если вас устраивают оба, исходите из ясности получаемого кода. Обычно использование `std::condition_variable` означает экономию четырех символов. Но обратите внимание, что из-за слоя изолирующей абстракции, предоставляемого типом `std::shared_lock`, фактический код для ожидания на `cv` под блокировкой чтения/записи почти идентичен коду ожидания на `cv` под обычным мьютексом. Вот версия на основе блокировки для чтения/записи:

```
bool ready = false;
std::shared_mutex ready_rwlock;
std::condition_variable_any cv;
std::thread thread_b([&]() {
  prep_work();
  {
    std::lock_guard lk(ready_rwlock);
    ready = true;
  }
  cv.notify_one();
  main_work();
```

```
});

// Ждать готовности потока B.
{
  std::shared_lock lk(ready_rwlock);
  while (!ready) {
    cv.wait(lk);
  }
}
// Поток B завершил подготовку к работе.
```

Это совершенно верный и максимально эффективный код. Однако ручная работа с блокировками и условными переменными почти так же опасна, как работа с обычными мьютексами или указателями. К счастью, у нас есть более удачные инструменты! Но о них мы поговорим в следующем разделе.

Обещания о будущем

Если прежде вам не приходилось сталкиваться с темами конкурентного программирования, последние несколько разделов в этой главе, вероятно, покажутся вам наиболее сложными. Мьютексы просты для понимания, потому что моделируют идею, знакомую по повседневной жизни: получение исключительного доступа к некоторому ресурсу путем запирания его на замок (установки блокировки). Блокировки для чтения/записи (`shared_mutex`) ненамного сложнее. Однако затем мы совершили настоящий скачок, рассмотрев условные переменные, которые трудно понять с первой попытки, потому что они моделируют не существительное (такое как «замок»), а скорее некоторую глагольную форму: «спать до тех пор, пока». К тому же свою лепту в сложности вносит их не очень удачное название.

Теперь мы продолжим наше путешествие по миру конкурентного программирования и исследуем тему, которая может быть незнакомой даже прошедшим курс обучения конкурентному программированию: механизмы *promise* и *future*[1].

В C++11 типы `std::promise<T>` и `std::future<T>` всегда соседствуют друг с другом. Пришедшие из языка Go могут представить пару promise/future как канал, в том смысле, что когда один поток вталкивает значение (типа `T`) на стороне «promise» этой пары, оно в конечном счете появится на стороне «future» (которая обычно находится в другом потоке выполнения). Пары promise/future похожи также на нестабильные червоточины (unstable wormholes): стоит только протолкнуть значение через червоточину, как она тут же схлопывается.

[1] Promise (обещание) и future (будущее) – два родственных механизма асинхронного выполнения. К сожалению, пока не существует устоявшихся русскоязычных терминов для их обозначения. Поэтому, чтобы не путать вас изобретенными терминами или кальками, я буду продолжать использовать термины «promise» и «future» без перевода. Дополнительную информацию можно найти в Википедии: https://ru.wikipedia.org/wiki/Futures_and_promises. – *Прим. перев.*

Можно сказать, что пара promise/future подобна направленной, перемещаемой червоточине однократного действия. Она «направленная», потому что втолкнуть данные можно только со стороны «promise», а извлечь их – только со стороны «future». Она «перемещаемая», потому что, владея концом червоточины, вы можете передавать его в функции и из функций и даже перемещать в другие потоки выполнения; туннель, соединяющий два конца, при этом не разрывается. А «однократная» она потому, что, втолкнув данные один раз со стороны «promise», вы не сможете втолкнуть туда что-либо повторно.

Другая метафора вытекает из их имен: фактически `std::future<T>` не является значением типа `T`, в некотором смысле это *будущее* (future) значение – оно даст вам доступ к значению `T` в некоторый момент в будущем, но «не прямо сейчас». (В этом смысле этот тип является неким подобием потокобезопасного `optional<T>`.) Объект `std::promise<T>`, в свою очередь, можно сравнить с пока не выполненным обещанием (promise), или долговой распиской. Владелец объекта promise *обещает* (promises) передать значение типа `T` в некоторый момент; если он этого не сделает, значит, он «нарушит обещание».

В общем случае использование пары promise/future начинается с создания `std::promise<T>`, где `T` – это тип данных, которые планируется передать через эту пару; затем создается конец «future» червоточины вызовом `p.get_future()`. Когда все будет готово к выполнению обещания, вызывается `p.set_value(v)`. Между тем, когда какой-то другой поток выполнения будет готов извлечь значение, он вызывает `f.get()`. Если вызов `f.get()` произойдет до того, как обещание будет выполнено, поток заблокируется до момента, пока не появится ожидаемое значение. С другой стороны, когда поток, удерживающий конец promise, вызовет `p.set_value(v)` и при этом никто не ожидает значения с другой стороны, ничего не произойдет; `set_value` просто запишет значение `v` в память, и другой поток сможет извлечь его в удобный для себя момент вызовом `f.get()`.

Давайте рассмотрим promise и future в работе!

```
std::promise<int> p1, p2;
std::future<int> f1 = p1.get_future();
std::future<int> f2 = p2.get_future();

// Если обещание будет выполнено раньше,
// тогда f.get() не заблокирует поток.
p1.set_value(42);
assert(f1.get() == 42);

// Если f.get() будет вызван раньше, тогда он
// заблокируется до момента, пока не будет вызван set_value()
// каким-то другим потоком.
std::thread t([&](){
  std::this_thread::sleep_for(100ms);
  p2.set_value(43);
});
```

```
auto start_time = std::chrono::system_clock::now();
assert(f2.get() == 43);
auto elapsed = std::chrono::system_clock::now() - start_time;
printf("f2.get() took %dms.\n", count_ms(elapsed));
t.join();
```

(Определение `count_ms` вы найдете в разделе «Специальные типы мьютексов» выше.)

Реализация `std::promise` в стандартной библиотеке имеет одну замечательную особенность – специализацию для типа `void`. На первый взгляд, идея `std::future<void>` может показаться странной – какая польза от червоточины, если единственный тип данных, который можно протолкнуть через нее, является типом без значения? Но на самом деле `future<void>` имеет высокую практическую ценность в ситуациях, когда нас интересует не столько конкретное значение, сколько сам сигнал. Например, `std::future<void>` можно использовать для реализации третьей версии нашего примера «ожидания запуска потока B»:

```
std::promise<void> ready_p;
std::future<void> ready_f = ready_p.get_future();

std::thread thread_b([&]() {
  prep_work();
  ready_p.set_value();
  main_work();
});

// Ждать готовности потока B.
ready_f.wait();
// Поток B завершил подготовку к работе.
```

Сравните эту версию с версией из раздела «Ожидание условия». Эта версия намного понятнее! В ней практически нет лишнего, шаблонного кода. Отправка сигнала «готовности потока B» и операция «ожидания готовности потока B» требуют всего по одной строке кода. Это определенно более удобный способ обмена сигналами между потоками. Существует также четвертый способ послать сигнал из одного потока целой группе других потоков, о котором рассказывается в разделе «Идентификация отдельных потоков и текущего потока».

Однако за удобство `std::future` приходится платить, платить распределением динамической памяти из кучи. Дело в том, что обоим механизмам, `promise` и `future`, требуется доступ к общему хранилищу, чтобы, например, значение 42, сохраненное на стороне promise, можно было извлечь на стороне future. (Доступ к такому хранилищу также регулируется посредством мьютекса и условной переменной, необходимых для синхронизации потоков. Мьютекс и условная переменная при этом не исчезают из нашего кода; они просто перемещаются на более низкий уровень абстракции, и нам не приходится беспокоиться о них.) То есть `promise` и `future` действуют как своеобразный

«дескриптор» этого общего состояния; но оба являются перемещаемыми типами, поэтому ни одному из них в действительности не требуется хранить общее состояние в виде члена данных. Они выделяют место в куче для общего состояния и хранят указатели на него; а поскольку общее состояние не должно освобождаться до уничтожения обоих дескрипторов, мы имеем дело с совместным владением, посредством чего-то, похожего на `shared_ptr` (см. главу 6 «Умные указатели»). Схематически `promise` и `future` выглядят, как показано на рис. 7.1.

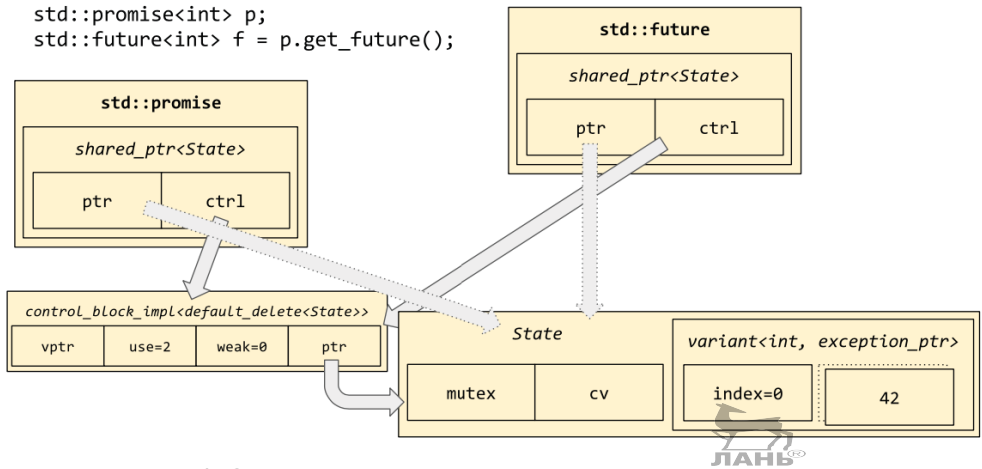


Рис. 7.1. Схематическое представление типов promise и future

Общее состояние на этой диаграмме размещается оператором `new`, если только не используется специализированная версия конструктора `std::promise` с нестандартным диспетчером памяти. Ниже показано, как использовать `std::promise` и `std::future` со своим диспетчером памяти:

```
MyAllocator myalloc{};
std::promise<int> p(std::allocator_arg, myalloc);
std::future<int> f = p.get_future();
```

`std::allocator_arg` определен в заголовке `<memory>`. Подробности относительно `MyAllocator` смотрите в главе 8 «Диспетчеры памяти».

Подготовка заданий для отложенного выполнения

Еще одна интересная особенность на предыдущей диаграмме, которую хотелось бы отметить, – общее состояние содержит не просто `optional<T>`; в действительности в нем хранится `variant<T, exception_ptr>` (см. главу 5 «Словарные типы»). Это означает, что в червоточину можно проталкивать не только данные типа `T`, но также исключения. Это особенно удобно, потому

что позволяет типу std::future<T> представлять все возможные результаты вызова функции с сигнатурой T(). Она может вернуть T; может возбудить исключение; а может вообще никогда ничего не вернуть. Аналогично вызов f.get() может вернуть T; возбудить исключение или (если поток, владеющий стороной promise, завис) вообще никогда не вернуть управления. Чтобы протолкнуть исключение через червоточину, следует использовать метод p.set_exception(ex), где ex – объект типа std::exception_ptr, который может быть возвращен из std::current_exception() внутри обработчика исключения.

Давайте возьмем функцию с сигнатурой T() и упакуем ее в тип std::future<T>:

```
template<class T>
class simple_packaged_task {
  std::function<T()> m_func;
  std::promise<T> m_promise;
public:
  template<class F>
  simple_packaged_task(const F& f) : m_func(f) {}

  auto get_future() { return m_promise.get_future(); }

  void operator()() {
    try {
      T result = m_func();
      m_promise.set_value(result);
    } catch (...) {
      m_promise.set_exception(std::current_exception());
    }
  }
};
```

Внешне этот класс напоминает тип std::packaged_task<R(A...)> из стандартной библиотеки, отличаясь только тем, что стандартный тип принимает аргументы и использует дополнительный уровень косвенности, гарантирующий возможность использования функторов, поддерживающих только перемещение. В главе 5 «Словарные типы» мы видели несколько решений, помогающих обойти ограничение std::function, не позволяющее хранить типы функций, поддерживающих только перемещение; к счастью, при работе с std::packaged_task эти обходные решения не нужны. С другой стороны, в реальной жизни вам едва ли придется использовать std::packaged_task. Он интересен в основном только как пример объединения promise, future и функций в удобные классы-типы с очень простым внешним интерфейсом. Остановимся на минуту: класс simple_packaged_task, представленный выше, использует стирание типа в std::function и имеет член std::promise, реализованный в терминах std::shared_ptr, который выполняет подсчет ссылок; общее состояние, на которое ссылается этот указатель с подсчетом ссылок, хранит мьютекс и переменную состояния. Как видите, в таком маленьком объеме оказалось

упаковано довольно много идей и технологий! Но, несмотря на это, `simple_packaged_task` имеет действительно очень простой интерфейс: его можно сконструировать на основе функции или лямбда-выражения, затем вызвать `pt.get_future()`, чтобы получить future, для которого, в свою очередь, можно вызвать `f.get()`; и в то же время вызвать `pt()` (возможно, из другого потока), чтобы фактически выполнить хранимую функцию и протолкнуть результат через червоточину в `f.get()`.

Если хранимая функция сгенерирует исключение, тогда `packaged_task` перехватит его (в потоке на стороне promise) и втолкнет его в червоточину. Затем, когда другой поток вызовет `f.get()` (или уже вызвал его и в данный момент ожидает возврата из `f.get()`), `f.get()` возбудит это исключение в потоке на стороне future. Иными словами, с помощью promise и future мы фактически «телепортируем» исключение между потоками. К сожалению, всестороннее обсуждение механизма телепортации, `std::exception_ptr`, выходит далеко за рамки этой книги. Если вы проектируете библиотеку, которая широко использует исключения, вам определенно стоит поближе познакомиться с `std::exception_ptr`.

Будущее механизма future

По аналогии с `std::shared_mutex`, версия `std::future` в стандартной библиотеке готова только наполовину. Гораздо более полная и полезная версия future появится, как я надеюсь, в C++20, но уже сейчас есть множество сторонних библиотек, реализующих лучшие черты будущего механизма future. К числу лучших в этом отношении относятся `boost::future` и `folly::Future`.

Основная проблема `std::future` – в вероятности «падения на землю» в потоке после каждого этапа в потенциально многоэтапных вычислениях. Рассмотрим следующий патологический случай использования `std::future`:

```
template<class T>
auto pf() {
  std::promise<T> p;
  std::future<T> f = p.get_future();
  return std::make_pair(std::move(p), std::move(f));
}

void test() {
  auto [p1, f1] = pf<Connection>();
  auto [p2, f2] = pf<Data>();
  auto [p3, f3] = pf<Data>();

  auto t1 = std::thread([p1 = std::move(p1)]() mutable {
    Connection conn = slowly_open_connection();
    p1.set_value(conn);
    // ОПАСНО: что, если slowly_open_connection возбудит исключение?
  });
```

```
    auto t2 = std::thread([p2 = std::move(p2)]() mutable {
      Data data = slowly_get_data_from_disk();
      p2.set_value(data);
    });
    auto t3 = std::thread(
    [p3 = std::move(p3), f1 = std::move(f1)]() mutable {
      Data data = slowly_get_data_from_connection(f1.get());
      p3.set_value(data);
    });
    bool success = (f2.get() == f3.get());

    assert(success);
  }
```

Обратите внимание на строку с комментарием ОПАСНО: все три потока содержат одну и ту же ошибку – они не перехватывают исключение и не вызывают `.set_exception()`. Исправить ошибку можно с помощью блока `try...catch`, как мы сделали это в `simple_packaged_task`, в предыдущем разделе; но, поскольку писать каждый раз один и тот же код слишком утомительно, в стандартную библиотеку была добавлена интересная функция-обертка `std::async()`, которая создает пару promise/future и запускает новый поток. Применив `std::async()`, можно получить намного более ясный код:

```
void test() {
  auto f1 = std::async(slowly_open_connection);
  auto f2 = std::async(slowly_get_data_from_disk);
  auto f3 = std::async([f1 = std::move(f1)]() mutable {
    return slowly_get_data_from_connection(f1.get());
    // Больше не опасно.
  });
  bool success = (f2.get() == f3.get());

  assert(success);
}
```

Однако это только эстетическая ясность; этот код чрезвычайно плохо сказывается на производительности и безопасности кода. Это *плохой* код!

Каждый раз, встречая `.get()` в этом коде, вы должны подумать: «Какая непростительная трата времени на переключение контекста!» И каждый раз, замечая запуск нового потока (явно или вызовом `async`), вы должны подумать: «Какая потрясающая возможность для операционной системы исчерпать потоки ядра и возбудить в конструкторе `std::thread` неожиданное исключение!» Вместо любого из предыдущих вариантов я предпочел бы написать что-нибудь этакое в стиле JavaScript:

```
void test() {
  auto f1 = my::async(slowly_open_connection);
  auto f2 = my::async(slowly_get_data_from_disk);
  auto f3 = f1.then([](Connection conn) {
    return slowly_get_data_from_connection(conn);
```

```
  });
  bool success = f2.get() == f3.get();

  assert(success);
}
```

Здесь `.get()` вызывается только в самом конце, когда уже ничего не нужно делать и остается только дождаться окончательного ответа; и здесь порождается на один поток меньше. Вместо запуска нового потока мы подключаем к `f1` «продолжение» до того, как тот завершит свое задание, чтобы сразу после этого поток на стороне promise смог перейти к выполнению продолжения (если первое задание `f1` возбудит исключение, вход в продолжение просто не будет выполнен. Библиотека должна также предоставить симметричный метод `f1.on_error(continuation)`, чтобы иметь возможность обработать исключение).

Нечто подобное уже имеется в библиотеке Boost, а библиотека Facebook Folly содержит особенно надежную и полноценную реализацию, даже лучшую, чем в Boost. Пока мы ждем, что C++20 улучшит ситуацию, я советую использовать Folly, если вы можете позволить себе дополнительные умственные нагрузки, связанные с интеграцией библиотеки в вашу систему сборки. Единственным преимуществом типа `std::future` является его стандартность; его можно использовать практически на любой платформе, не заботясь о загрузке дополнительных библиотек, путях поиска подключаемых файлов и лицензионных соглашениях.

Поговорим о потоках...

На протяжении всей этой главы мы использовали слово «поток» (или «поток выполнения»), даже не определив, что оно означает в действительности; и вы, вероятно, заметили, что во многих примерах многопоточного кода используется класс типа `std::thread` и пространство имен `std::this_thread` без каких-либо пояснений. Мы сосредоточились на том, *как* синхронизировать работу разных потоков выполнения, но до сих пор умалчивали о том, *кто* или *что* выполняется!

Иначе говоря: когда выполнение достигает выражения `mtx.lock()`, где `mtx` – это заблокированный мьютекс, семантика `std::mutex` говорит, что текущий поток выполнения должен заблокироваться и ждать. Что происходит, пока этот поток заблокирован? Наша программа на C++ все еще «отвечает» на происходящее, но очевидно, что *этот заблокированный код на C++* больше не выполняется; тогда кто или что выполняется? Ответ прост: другой поток. Мы указываем на существование других потоков и код, который они должны выполнить, используя класс `std::thread` из стандартной библиотеки, который определен в заголовке `<thread>`.

Чтобы запустить новый поток выполнения, нужно лишь сконструировать объект типа `std::thread`, передав конструктору единственный аргумент:

лямбда-выражение или функцию с кодом, который должен выполнить новый поток. (Технически можно передать несколько аргументов; все остальные аргументы, следующие за первым, будут переданы в указанную функцию в виде параметров, после развертывания с `reference_wrapper`, как описывалось в главе 5 «Словарные типы». С появлением поддержки лямбда-выражений в C++11 дополнительные аргументы конструктора `thread` стали не нужны и даже нежелательны; я советую избегать их.)

Новый поток запускается немедленно; если понадобится выполнить «запуск с задержкой», вам придется самим реализовать это, используя один из приемов синхронизации, показанных в разделе «Ожидание условия» или в разделе «Идентификация отдельных потоков и текущего потока».

Новый поток выполнит переданный ему код и, достигнув конца лямбда-выражения или функции, станет «доступным для присоединения». Это очень похоже на происходящее с `std::future`, когда он становится «готовым к чтению»: поток завершает вычисления и готов передать полученные результаты. Так же как в случае с `std::future<void>`, результат вычислений «не имеет значения»; но важен сам факт завершения вычислений!

В отличие от `std::future<void>`, однако нельзя уничтожить объект `std::thread` не получив результат без значения. По умолчанию, если уничтожить любой новый поток, не обработав его результат, деструктор вызовет `std::terminate` и тем самым завершит всю программу. Чтобы избежать такой судьбы, нужно указать на поток и подтвердить его завершение – «Хорошая работа, поток, молодец!» – вызвав функцию-член `t.join()`. Как вариант, если вы не ожидаете завершения потока (например, если фоновый поток выполняет бесконечный цикл) или вас не интересует его результат (например, если это была короткоживущая задача, действующая по принципу «запустил и забыл»), можно перевести поток в фоновый режим – «Все, уходи, я больше не желаю ничего знать о тебе!» – вызовом `t.detach()`.

Вот несколько законченных примеров использования `std::thread`:

```
using namespace std::literals; // для "ms"

std::thread a([](){
  puts("Thread A says hello ~0ms");
  std::this_thread::sleep_for(10ms);
  puts("Thread A says goodbye ~10ms");
});

std::thread b([](){
  puts("Thread B says hello ~0ms");
  std::this_thread::sleep_for(20ms);
  puts("Thread B says goodbye ~20ms");
});

puts("The main thread says hello ~0ms");
a.join();   // ждать завершения потока A
```

```
b.detach(); // не ждать завершения потока B
puts("The main thread says goodbye ~10ms");
```

Идентификация отдельных потоков и текущего потока

Объекты типа `std::thread`, подобно всем другим типам, описанным в этой главе, не поддерживают `operator==`. Вы не сможете напрямую спросить: «Эти два объекта потоков – одно и то же?» Это также означает, что объекты `std::thread` нельзя использовать в качестве ключей в ассоциативных контейнерах, таких как `std::map` или `std::unordered_map`. Зато о равенстве *можно* спросить косвенно, обратившись к *механизму идентификации потоков*.

Функция-член `t.get_id()` возвращает уникальный идентификатор типа `std::thread::id`, который, хотя технически является классом типа, во многом ведет себя как целочисленный тип. Идентификаторы потоков можно сравнивать с помощью операторов `<` и `==` и использовать в качестве ключей в ассоциативных контейнерах. Другая ценная особенность идентификаторов потоков – возможность их копировать, в отличие от самих объектов `std::thread`, которые поддерживают только перемещение. Не забывайте, что каждый объект `std::thread` фактически представляет действующий поток выполнения; если бы можно было копировать объекты потоков, тогда должна была бы иметься возможность «скопировать» сами потоки выполнения, что лишено всякого смысла и могло бы породить некоторые интересные ошибки!

Третья ценная особенность `std::thread::id` – возможность получить идентификатор *текущего* и даже главного потока. Внутри потока нельзя сказать: «Пожалуйста, дай мне объект `std::thread`, который управляет этим потоком». (Это можно сравнить с трюком `std::enable_shared_from_this<T>` из главы 6 «Умные указатели», но, как мы видели, такой трюк требует поддержки со стороны библиотеки, создающей управляемые ресурсы, каковым в данном случае оказался бы конструктор `std::thread`.) Главный поток – поток, в котором начинает выполняться функция `main`, – вообще не имеет соответствующего объекта `std::thread`, но он имеет идентификатор!

Наконец, идентификаторы потоков можно преобразовывать некоторым способом, зависящим от реализации, в строковое представление, которое гарантирует уникальность, то есть условие `to_string(id1) == to_string(id2)` выполнится тогда и только тогда, когда `id1 == id2`. К сожалению, это строковое представление доступно только посредством оператора потока (см. главу 9 «Потоки ввода/вывода»). Если вы захотите использовать синтаксис `to_string(id1)`, вам придется написать свою функцию-обертку, например:

```
std::string to_string(std::thread::id id)
{
  std::ostringstream o;
  o << id;
```

```
  return o.str();
}
```

Получить идентификатор текущего потока (включая главный, если он является текущим) можно вызовом свободной функции `std::this_thread::get_id()`. Обратите внимание на синтаксис! `std::thread` – это имя класса, но `std::this_thread` – это имя пространства имен. В этом пространстве имен определено несколько свободных функций (не связанных ни с какими экземплярами классов), управляющих текущим потоком. Одна из таких функций – `get_id()`. Ее имя напоминает `std::thread::get_id()`, но в действительности это совершенно иная функция: `thread::get_id()` – это функция-член, а `this_thread::get_id()` – свободная функция.

Используя два идентификатора, можно определить, например, какой из потоков в списке является текущим:

```
std::mutex ready;
std::unique_lock lk(ready);
std::vector<std::thread> threads;

auto task = [&](){
  // Подождать, пока главный поток не будет готов.
  (void)std::lock_guard(ready);
  // Продолжаем. найти свой идентификатор в векторе.
  auto my_id = std::this_thread::get_id();
  auto iter = std::find_if(
    threads.begin(), threads.end(),
    [=](const std::thread& t) {
      return t.get_id() == my_id;
    }
  );
  printf("Thread %s %s in the list.\n",
    to_string(my_id).c_str(),
    iter != threads.end() ? "is" : "is not");
};

std::vector<std::thread> others;
for (int i = 0; i < 10; ++i) {
  std::thread t(task);
  if (i % 2) {
    threads.push_back(std::move(t));
  } else {
    others.push_back(std::move(t));
  }
}

// Позволить всем потокам продолжить выполнение.
ready.unlock();

// Присоединить все потоки.
for (std::thread& t : threads) t.join();
for (std::thread& t : others) t.join();
```

Чего нельзя сделать, так это пойти в обратном направлении – нельзя реконструировать объект `std::thread`, соответствующий заданному `std::thread::id`. Если бы это было возможно, тогда вы смогли бы получить два разных объекта, представляющих один поток выполнения: оригинальный `std::thread`, где бы он ни находился, и воссозданный вами из идентификатора. Но вы никогда не сможете получить два объекта `std::thread`, управляющих одним и тем же потоком.

В пространстве имен `std::this_thread` имеются еще две свободные функции: `std::this_thread::sleep_for(duration)`, которую я часто использовал в этой главе, и `std::this_thread::yield()`, фактически имеющая тот же смысл, что `sleep_for(0ms)`, – она сообщает среде времени выполнения, что было бы неплохо прямо сейчас переключить контекст на другой поток, но не назначает никакой задержки в выполнении для текущего потока.

Исчерпание потоков и std::async

В разделе «Будущее механизма future» мы познакомились с `std::async` – простой оберткой вокруг конструктора потока, результат работы которого захватывается в `std::future`. Ее реализация в общих чертах выглядит примерно так:

```
template<class F>
auto async(F&& func) {
  using ResultType = std::invoke_result_t<std::decay_t<F>>;
  using PromiseType = std::promise<ResultType>;
  using FutureType = std::future<ResultType>;

  PromiseType promise;
  FutureType future = promise.get_future();
  auto t = std::thread([
    func = std::forward<F>(func),
    promise = std::move(promise)
  ]() mutable {
    try {
      ResultType result = func();
      promise.set_value(result);
    } catch (...) {
      promise.set_exception(std::current_exception());
    }
  });

  // Эта особенность не реализуется
  // за пределами библиотеки, но async содержит ее.
  // future.on_destruction([t = std::move(t)]() {
  //   t.join();
  // });
  return future;
}
```

Обратите внимание на закомментированные строки, определяющие особое поведение «при уничтожении» `std::future`, возвращаемого из `std::async`. Это странное и неуклюжее поведение реализации `std::async` в стандартной библиотеке является веской причиной, чтобы избегать использования `std::async` в своем коде: экземпляры future, возвращаемые из `std::async`, имеют деструкторы, вызывающие `.join()` своих потоков! Это означает, что их деструкторы могут заблокироваться и задача определенно не будет «выполняться в фоновом режиме», как вы могли бы ожидать. Если вы вызываете `std::async` и не присваиваете возвращаемый future переменной, возвращаемое значение тут же будет уничтожено, а это значит, что строка, не содержащая ничего, кроме вызова `std::async`, фактически выполнит указанную функцию синхронно:

```
template<class F>
void fire_and_forget_wrong(const F& f) {
  // ОШИБКА! Запустит f в другом потоке, но заблокируется.
  std::async(f);
}

template<class F>
void fire_and_forget_better(const F& f) {
  // ЛУЧШЕ! Запустит f в другом потоке и не заблокирует его.
  std::thread(f).detach();
}
```

Главная причина введения этого ограничения, по всей видимости, объясняется беспокойством, что если `std::async` будет запускать фоновые потоки как обычно, это подтолкнет людей к злоупотреблению `std::async` и повлечет появление ошибок, связанных с искажением ссылок, как в следующем примере:

```
int test() {
  int i = 0;
  auto future = std::async([&]() {
    i += 1;
  });
  // представим, что мы не вызываем здесь f.wait()
  return i;
}
```

Если не дожидаться появления результата в future, функция `test()` могла бы вернуть управление вызывающему коду еще до его запуска; в этом случае, когда новый поток наконец запустится, он попытается увеличить переменную `i` на стеке, которой больше не существует. Поэтому, чтобы не подвергать риску людей, способных написать такой ошибочный код, комитет по стандартизации решил, что `std::async` должна возвращать future со специальным «волшебным» деструктором, который автоматически присоединяет свой поток.

Как бы то ни было, но злоупотребление `std::async` представляет проблему еще по целому ряду причин. Самая существенная из них – во всех популярных

операционных системах тип `std::thread` представляет поток ядра, то есть поток, работой которого управляет ядро операционной системы. Поскольку операционные системы имеют ограниченные ресурсы, которые можно направить на управление потоками, количество потоков, доступных одному процессу, значительно ограничено: часто не более нескольких десятков тысяч. Если использовать `std::async` в роли диспетчера потоков, порождающего новый поток `std::thread`, когда появляется новая задача, можно быстро столкнуться с ошибкой исчерпания потоков ядра. В этой ситуации конструктор `std::thread` начнет возбуждать исключения типа `std::system_error`, обычно с текстом `Resource temporarily unavailable` (Ресурс временно недоступен).

Создание своего пула потоков

Если использовать `std::async` для создания потока, когда появляется новая задача, есть риск превысить число потоков ядра, доступных процессу. Более удачный способ организации параллельного выполнения задач – использовать *пул потоков*, небольшое количество «рабочих потоков», единственной работой которых является выполнение задач, передаваемых программистом. Если число задач превысит число рабочих потоков, *избыточные* задачи помещаются в *рабочую очередь*. Завершив выполнение очередной задачи, рабочий поток проверяет рабочую очередь и извлекает из нее следующую задачу.

Это хорошо известная идея, но она до сих пор не включена в стандартную библиотеку. Однако можно объединить идеи, представленные выше в этой главе, и реализовать свой пул потоков. Далее я расскажу о простой, пусть не самой производительной, но зато потокобезопасной и правильной во всех смыслах реализации пула потоков. После этого мы обсудим некоторые оптимизации, которые помогут повысить производительность.

Начнем с члена-данных. Далее мы будем следовать правилу, согласно которому все данные, управляемые мьютексом, должны находиться рядом, в одном видимом пространстве; в данном случае используется вложенная структура. Также мы будем использовать `std::packaged_task<void()>` как тип функции, допускающей только возможность перемещения; если в вашем проекте уже имеется такой тип функции, можете использовать его. Если у вас нет такого типа, я предлагаю воспользоваться типом `folly::Function` из библиотеки Folly или типом `fu2::unique_function`, разработанным Денисом Бланком (Denis Blank):

```
class ThreadPool {
  using UniqueFunction = std::packaged_task<void()>;
  struct {
    std::mutex mtx;
    std::queue<UniqueFunction> work_queue;
    bool aborting = false;
  } m_state;
  std::vector<std::thread> m_workers;
  std::condition_variable m_cv;
```

Переменная work_queue будет хранить задания, поступающие извне. Переменная-член m_state.aborting получит значение true, когда придет время всем рабочим потокам остановить работу и «отправиться домой на отдых». m_workers хранит сами рабочие потоки; а m_state.mtx и m_cv служат для синхронизации. (Большую часть времени, в отсутствие заданий, рабочие потоки будут спать. Когда появится новое задание и нам понадобится разбудить рабочие потоки, мы сможем сделать это, уведомив их посредством m_cv.)

Конструктор ThreadPool создает рабочие потоки и заполняет вектор m_workers. Каждый рабочий поток будет выполнять функцию-член this->worker_loop(), которую вы увидите далее:

```
public:
  ThreadPool(int size) {
    for (int i=0; i < size; ++i) {
      m_workers.emplace_back([this]() { worker_loop(); });
    }
  }
```

Как обещалось, деструктор присваивает члену m_state.aborting значение true и затем ждет, пока все потоки заметят изменение и прекратят выполнение. Обратите внимание, что чтение m_state.aborting производится под защитой m_state.mtx; мы стараемся следовать правилам личной гигиены, чтобы избежать появления жучков в коде!

```
~ThreadPool() {
  if (std::lock_guard lk(m_state.mtx); true) {
    m_state.aborting = true;
  }
  m_cv.notify_all();
  for (std::thread& t : m_workers) {
    t.join();
  }
}
```

Теперь посмотрим, как добавлять задания в рабочую очередь. (Здесь не показано, как задания извлекаются из очереди, – эту операцию вы увидите в функции-члене worker_loop.) Это просто: доступ к m_state должен производиться только под защитой мьютекса, а после добавления задания в очередь нужно вызвать m_cv.notify_one(), чтобы разбудить один из свободных рабочих потоков для выполнения задания:

```
void enqueue_task(UniqueFunction task) {
  if (std::lock_guard lk(m_state.mtx); true) {
    m_state.work_queue.push(std::move(task));
  }
  m_cv.notify_one();
}
```

А теперь сам рабочий цикл. Это – функция-член, которую выполняют все рабочие потоки:

```
private:
  void worker_loop() {
    while (true) {
      std::unique_lock lk(m_state.mtx);
      while (m_state.work_queue.empty() && !m_state.aborting) {
        m_cv.wait(lk);
      }
      if (m_state.aborting) break;
      // Вытолкнуть следующее задание под зажитой мьютекса.
      assert(!m_state.work_queue.empty());
      UniqueFunction task = std::move(m_state.work_queue.front());
      m_state.work_queue.pop();

      lk.unlock();
      // Выполнить задание. На это может потребоваться некоторое время.
      task();
      // После выполнения задания проверить наличие следующего в очереди.
    }
  }
```

Обратите внимание на неизбежный цикл вокруг `m_cv.wait(lk)` и на то, что по соображениям гигиены доступ к `m_state` осуществляется только под защитой мьютекса. Также отметьте, что перед фактическим выполнением задания мы сначала освобождаем мьютекс; это гарантирует, что блокировка не будет удерживаться излишне долго, пока выполняется задание. Если не освободить блокировку, тогда никакой другой рабочий поток не смог бы получить ее, чтобы извлечь следующее задание из очереди – мы фактически исключили бы возможность параллельного выполнения нескольких заданий. Кроме того, если не освободить блокировку перед началом выполнения задания, тогда при попытке добавить новое задание в очередь (для чего требуется приобрести блокировку) это задание вызвало бы состояние взаимоблокировки с программой и сделало бы невозможным продолжение работы. Это особый случай более общего правила: никогда не вызывать пользовательскую функцию, удерживая блокировку.

Наконец, завершим класс `ThreadPool` реализацией безопасной версии `async`. Наша версия позволит вызвать `tp.async(f)` для любой функции `f` без аргументов, и так же, как `std::async`, будем возвращать `std::future`, с помощью которого вызывающий код получит результат `f`, когда тот будет готов. В отличие от future, возвращаемого из `std::async`, наш future можно безопасно игнорировать, если вызывающий код решит не ждать результата, задание при этом останется в очереди и в конечном итоге будет выполнено, а результат выполнения будет просто отброшен:

```
public:
  template<class F>
```

```
  auto async(F&& func) {
    using ResultType = std::invoke_result_t<std::decay_t<F>>;

    std::packaged_task<ResultType()> pt(std::forward<F>(func));
    std::future<ResultType> future = pt.get_future();

    UniqueFunction task(
      [pt = std::move(pt)]() mutable { pt(); }
    );

    enqueue_task(std::move(task));

    // Вернуть future для извлечения результата.
    return future;
  }
}; // class ThreadPool
```

Используя класс `ThreadPool`, можно написать, например, такой код, который создает 60 000 заданий:

```
void test() {
  std::atomic<int> sum(0);
  ThreadPool tp(4);
  std::vector<std::future<int>> futures;
  for (int i=0; i < 60000; ++i) {
    auto f = tp.async([i, &sum](){
      sum += i;
      return i;
    });
    futures.push_back(std::move(f));
  }
  assert(futures[42].get() == 42);
  assert(903 <= sum && sum <= 1799970000);
}
```

Можно было бы попробовать реализовать то же самое с помощью `std::async`, но мы наверняка столкнулись бы с исчерпанием потоков ядра. Предыдущий пример использует только четыре потока ядра, как подсказывает параметр конструктора `ThreadPool`.

Запустив этот код, вы увидите в стандартном выводе числа от 0 до 42, не обязательно следующие по порядку. Мы знаем, что число 42 точно должно появиться, потому что функция определенно ждет готовности `futures[42]` перед завершением, а все предшествующие числа должны быть выведены, потому что соответствующие им задания помещаются в очередь перед заданием с числом 42. Числа от 43 до 59 999 могут появиться, а могут не появиться, в зависимости от особенностей работы планировщика; потому что как только задание 42 завершится, функция `test` завершается, что вызывает уничтожение пула потоков. Деструктор пула, как мы уже видели, уведомляет свои рабочие потоки о необходимости прекратить работу после завершения текущих зада-

ний. То есть весьма вероятно, что вы увидите в стандартном выводе чуть больше чисел, но после этого все рабочие потоки остановятся и остальные задания будут отброшены.

Конечно, если вы пожелаете заблокировать деструктор класса `ThreadPool` до выполнения всех заданий, имеющихся в очереди, вам придется изменить код деструктора. Но обычно деструктор вызывается программой, потому что она завершает работу, например, когда вы нажимаете комбинацию **Ctrl+C**. В таком случае предпочтительнее прервать выполнение как можно раньше, не пытаясь опустошить очередь обычным порядком. Лично я предпочел бы добавить функцию-член `tp.wait_for_all_enqueued_tasks()`, чтобы пользователь пула имел возможность выбирать, выполнить ли все задания в очереди или просто сбросить их.

Оптимизация производительности пула потоков

Самое узкое место в нашей реализации `ThreadPool` – состязание рабочих потоков за обладание одним и тем же мьютексом, `this->m_state.mtx`. Причина такого состязания объясняется тем, что данный мьютекс защищает очередь `this->m_state.work_queue`, к которой вынуждены обращаться все рабочие потоки, чтобы получить следующее задание. Поэтому уменьшить конкуренцию и повысить производительность программы можно, если найти способ распределения заданий между рабочими потоками без использования общей центральной очереди.

Простейшее решение – дать каждому рабочему потоку свой «список дел»; то есть заменить единую очередь `std::queue<Task>` вектором `std::vector<std::queue<Task>>`, содержащим по одному элементу для каждого рабочего потока. Конечно, тогда нам придется также добавить `std::vector<std::mutex>` с мьютексом для каждой отдельной рабочей очереди. Функция `enqueue_task` могла бы распределять задания по рабочим очередям друг за другом (используя атомарный счетчик `std::atomic<int>` для одновременной обработки нескольких очередей).

В качестве альтернативы можно использовать счетчик `thread_local` для каждого отдельного потока, если вам посчастливилось работать на платформе, поддерживающей ключевое слово `thread_local` из стандарта C++11. На платформах x86-64 POSIX доступ к переменной `thread_local` осуществляется почти так же быстро, как доступ к обычной глобальной переменной; все сложности, связанные с подготовкой локальных переменных потоков, скрыты за кулисами и преодолеваются только один раз, при запуске нового потока. Однако поскольку эти сложности имеют место и должны поддерживаться средой выполнения, многие платформы пока не поддерживают спецификатор класса хранения `thread_local`. (На тех, где он поддерживается, объявление `thread_local int x` почти эквивалентно объявлению `static int x`, за исключением

того, что когда код обращается к x по имени, фактический адрес x в памяти зависит от std::this_thread::get_id(). В принципе, где-то за кулисами существует целый массив x, индексируемый идентификаторами потоков и заполняемый средой выполнения C++ при создании и уничтожении потоков.)

Следующим значительным усовершенствованием нашего класса ThreadPool могла бы стать организация «кражи заданий»: теперь, когда каждый рабочий поток имеет свой список дел, может так получиться, случайно или намеренно, что один рабочий поток окажется перегружен, а другие в это время будут простаивать. В таком случае хотелось бы, чтобы простаивающие рабочие потоки проверяли очереди занятых потоков и «крали» у них задания, если это возможно. При этом снова возникает состязание за обладание блокировками среди рабочих потоков, но только когда возникает неэффективность, вызванная несправедливым распределением заданий, – неэффективность, которую мы пытаемся *исправить* посредством кражи заданий.

Реализацию отдельных рабочих очередей и приема кражи заданий я оставляю читателям как самостоятельное упражнение; и надеюсь, что после знакомства с базовой реализацией ThreadPool вас не испугает идея изменить его и включить эти дополнительные возможности.

Конечно, существуют также классы пулов потоков, написанные профессионалами. Boost.Asio, например, содержит один такой класс, и Asio находится на пути включения в стандарт, возможно, даже в C++20. Если задействовать Boost.Asio, наш класс ThreadPool мог бы выглядеть как-то так:

```
class ThreadPool {
  boost::thread_group m_workers;
  boost::asio::io_service m_io;
  boost::asio::io_service::work m_work;
public:
  ThreadPool(int size) : m_work(m_io) {
    for (int i=0; i < size; ++i) {
      m_workers.create_thread([&](){ m_io.run(); });
    }
  }

  template<class F>
  void enqueue_task(F&& func) {
    m_io.post(std::forward<F>(func));
  }

  ~ThreadPool() {
    m_io.stop();
    m_workers.join_all();
  }
};
```

Описание Boost.Asio, как вы понимаете, выходит далеко за рамки этой книги.

Каждый раз, используя пул потоков, будьте внимательны и в задачах, добавляемых в очередь, никогда не приостанавливайте выполнение в ожидании

условий, зависящих от других заданий в том же пуле потоков. Классическим примером может служить задача A, приостанавливающаяся на условной переменной в ожидании, что некоторая задача B позднее изменит условную переменную. Если создать пул потоков `ThreadPool` с размером 4, добавить в очередь четыре копии задания A, а потом четыре копии здания B, обнаружится, что задание B не может запуститься – четыре потока в пуле заняты четырьмя копиями задания A и все они приостановились в ожидании сигнала, который некому послать! «Обработка» этого случая равносильна написанию собственной библиотеки поддержки многопоточного выполнения; если вы не хотите заниматься этой работой, тогда вам остается только проявить здоровую осторожность, чтобы не допустить подобного сценария.

Итоги

Многопоточность – сложная тема, полная нюансов и ловушек, что очевидно, только когда оглядываешься назад. В этой главе мы узнали:

Что спецификатор `volatile` хорош для работы с аппаратурой, но его недостаточно для безопасности в многопоточном окружении. Что `std::atomic<T>` для скалярного типа `T` (умещающегося в регистре процессора) – хороший способ доступа к общим данным без блокировок и без опасности попасть в состояние гонки. Наиболее важной атомарной операцией является операция «сравнить и поменять», которая на языке C++ записывается как `compare_exchange_weak`.

Чтобы заставить потоки по очереди обращаться к общим неатомарным данным, мы обычно используем `std::mutex`. Всегда запирайте мьютексы с помощью RAII-класса, такого как `std::unique_lock<M>`. Помните: даже при том, что автоматическое определение шаблонного аргумента в C++17 позволяет нам опускать `<M>` в именах этих шаблонов, это всего лишь синтаксическое соглашение; они продолжают оставаться шаблонными классами.

В своих программах всегда ясно показывайте, какие данные контролирует каждый мьютекс. Для этого прекрасно подходят вложенные структуры.

`std::condition_variable` позволяет «дождаться» некоторого условия. Если условие может наступить только один раз, как, например, завершение потока, тогда для этой цели вместо условной переменной, вероятно, лучше использовать пару promise/future. Если условие может наступать снова и снова, подумайте – можно ли вашу задачу сформулировать в терминах шаблона рабочей очереди.

`std::thread` воплощает идею потока выполнения. «Текущим потоком» нельзя напрямую манипулировать посредством объекта `std::thread`, тем не менее некоторые операции все же доступны в виде ограниченного набора свободных функций в пространстве имен `std::this_thread`. Наиболее важными из них являются операции `sleep_for` и `get_id`. Каждый объект `std::thread` всегда должен присоединяться или отсоединяться перед уничтожением. Отсо-

единение может пригодиться только для фоновых потоков, которые не требуют явного уничтожения.

Стандартная функция `std::async` принимает функцию или лямбда-выражение для выполнения в некотором другом потоке и возвращает объект `std::future`, который получает результат, когда функция завершит выполнение. Стандартная реализация `std::async`, однако, содержит фатальную ошибку (деструктор присоединяет поток; возможно исчерпание потоков ядра) и не должна использоваться в промышленном коде, поэтому для решения проблем конкуренции предпочтительнее использовать механизм future. Используйте реализацию promise/future, поддерживающую метод `.then`. Лучшей считается реализация в библиотеке Folly.

Многопоточность – сложная тема, полная нюансов и подводных камней, которые становятся очевидными только после накопления опыта.

Глава 8

Диспетчеры памяти

В предыдущих главах мы видели, что C++ испытывает любовь и ненависть к размещению данных в динамической памяти.

С одной стороны, использование динамической памяти является явным признаком «плохого кода»; погоня за указателями может ухудшить производительность программы, куча может неожиданно исчерпаться (и привести к исключению типа `std::bad_alloc`), а ручное управление памятью настолько сложно, что в C++11 было введено несколько типов «умных указателей» для устранения сложностей (см. главу 6 «Умные указатели»). В версии C++, появившиеся после 2011 года, было добавлено большое количество алгебраических типов данных, не использующих динамическую память, таких как `tuple`, `optional` и `variant` (см. главу 5 «Словарные типы»), которые могут выражать владение, не касаясь кучи.

С другой стороны, новые типы умных указателей позволяют эффективно справляться со сложностями, связанными с управлением памятью; в современном C++ можно эффективно выделять и освобождать память вообще без использования `new` и `delete`, не опасаясь утечек памяти. И распределение из кучи используется «за кулисами» многими новыми возможностями, появившимися в C++ (`any`, `function`, `promise`), потому что они пользуются многими старыми механизмами (`stable_partition`, `vector`).

Возникает конфликт: как можно использовать новые (и старые) механизмы, которые зависят от распределения памяти из кучи, если при этом утверждается, что хороший код на C++ не пользуется динамической памятью кучи?

Обычно предпочтение следует отдавать стандартным инструментам C++. Если вам нужен массив элементов с возможностью динамического изменения размеров, вы по умолчанию должны использовать `std::vector`, если при этом не возникают неприемлемые издержки производительности. Но существует также другая группа программистов, создающих программное обеспечение для выполнения в окружениях с очень ограниченными ресурсами, как, например, программное обеспечение управления полетом, которые должны избегать использования динамической памяти по одной простой причине: в их окружениях нет «кучи»! В таких встраиваемых системах вся раскладка памяти должна быть известна уже на этапе компиляции. В некоторых из таких программ просто не используются алгоритмы, выполняющие распределение

памяти из кучи, – вы никогда не столкнетесь с проблемой исчерпания ресурса, если никогда не прибегаете к механизмам динамического распределения этого ресурса! В других таких программах используются алгоритмы, выполняющие распределение памяти из кучи, но они требуют, чтобы «куча» была явно представлена в них (например, в виде большого массива `char` и функций, «резервирующих» и «возвращающих» последовательные блоки памяти в этот массив).

Было бы очень неудобно, если бы подобные программы не могли использовать механизмы языка C++, такие как `std::vector` и `std::any`. Поэтому, начиная со стандарта, вышедшего в 1998, стандартная библиотека предоставляет механизм поддержки *сторонних диспетчеров памяти*. Когда тип или алгоритм поддерживает возможность использовать такой сторонний диспетчер памяти, он дает программисту возможность явно указать, как этот тип или алгоритм будет резервировать и возвращать динамическую память. Это «как» воплощается в объект, известный как *диспетчер памяти*, или *распределитель* (allocator).

В этой главе вы узнаете:

- как определяются понятия «диспетчер памяти» и «ресурс памяти»;
- как создать свой ресурс памяти для выделения блоков из статического буфера;
- как определить свои контейнеры, использующие нестандартный диспетчер памяти;
- стандартные типы ресурсов памяти из пространства имен `std::pmr` и их подводные камни;
- многие из странных особенностей модели диспетчеров памяти в C++11, предназначенных исключительно для поддержки `scoped_allocator_adaptor`;
- что делает тип «причудливым указателем» и где такой тип может пригодиться.

Диспетчер памяти обслуживает ресурс памяти

Читая эту главу, вы должны постоянно помнить о различиях между двумя фундаментальными понятиями, которые я называю *ресурсом памяти* (memory resource) и *диспетчером памяти*, или распределителем (allocator). Ресурс памяти (это название взято из собственной терминологии стандарта – возможно, вам более естественным покажется название «куча») – это долгоживущий объект, который может выделять фрагменты памяти по запросу (обычно из более крупного блока памяти, который принадлежит самому ресурсу памяти). Ресурсы памяти обладают классической объектно-ориентированной семантикой (см. главу 1 «Классический полиморфизм и обобщенное программирование»):

вы создаете ресурс памяти один раз и никогда не копируете и не перемещаете его, и равенство ресурсов памяти обычно определяется идентичностью объектов. С другой стороны, диспетчер памяти (распределитель) – короткоживущий дескриптор, указывающий на ресурс памяти. Диспетчеры памяти проявляют семантику указателей: их можно копировать, перемещать и вообще оперировать ими почти без ограничений, и равенство диспетчеров памяти обычно определяется равенством указателей на ресурс памяти. Вместо «диспетчер памяти указывает на такой-то ресурс памяти» можно также сказать, что «диспетчер памяти встроен в ресурс памяти»; эти фразы взаимозаменяемы.

Когда я буду рассказывать о «ресурсах памяти» и «диспетчерах памяти» в этой главе, я также поведаю об идеях, предшествовавших им. В стандартной библиотеке имеется также несколько типов с именами `memory_resource` и `allocator`; всякий раз, когда речь будет заходить об этих типах, я буду использовать оформление моноширинным шрифтом. Пусть это не сбивает вас с толку. Аналогичная ситуация возникала в главе 2 «Итераторы и диапазоны», где я рассказывал об «итераторах» и о типе `std::iterator`. Конечно, тогда было проще, потому что я говорил, что не следует использовать `std::iterator` в своих программах; ему нет места в хорошем коде на C++. В этой главе вы узнаете, что использование `std::pmr::memory_resource` *вполне оправданно* в некоторых программах на C++!

Несмотря на то что я назвал диспетчера памяти дескриптором, «указывающим на» ресурс памяти, имейте в виду, что иногда ресурс памяти является глобальным синглтоном – ярким примером такого синглтона может служить глобальная куча, управление которой реализуется глобальными операторами `new` и `delete`. Подобно тому, как лямбда-выражение, «замыкающее» глобальную переменную, в действительности ничего не замыкает, диспетчер, встроенный в глобальную кучу, не нуждается ни в каком состоянии. Фактически `std::allocator<T>` – именно такой тип диспетчера без состояния, но не будем забегать вперед!

Еще раз об интерфейсах и понятиях

В главе 1 «Классический полиморфизм и обобщенное программирование» рассказывалось, что C++ предлагает два, в основном несовместимых, способа интерпретации полиморфизма. Статический полиморфизм времени компиляции называется *обобщенным программированием*; он опирается на выражение полиморфного интерфейса как идеи со множеством возможных *моделей*, а код, взаимодействующий с интерфейсом, выражается в терминах *шаблонов*. Динамический полиморфизм времени выполнения называется *классическим полиморфизмом*; он опирается на выражение полиморфного интерфейса в виде *базового класса* со множеством возможных *производных классов*, а код, взаимодействующий с интерфейсом, выражается в терминах вызовов *виртуальных методов*.

В этой главе мы в первый (и в последний) раз по-настоящему близко столкнемся с обобщенным программированием. Понять диспетчеры памяти в C++

невозможно, не подразумевая два понятия сразу: с одной стороны, *понятие* `Allocator`, определяющее интерфейс, и с другой – некоторая конкретная *модель*, такая как `std::allocator`, реализующая поведение, соответствующее понятию `Allocator`.

Самое примечательное, что `Allocator` в действительности является семейством понятий! Точнее было бы называть его семейством понятий `Allocator<T>`; например, `Allocator<int>` определяет понятие «диспетчера памяти, распределяющего объекты `int`», `Allocator<char>` – «диспетчер памяти, распределяющий объекты `char`», и т. д. И например, конкретный класс `std::allocator<int>` является моделью понятия `Allocator<int>`, но он не является моделью для `Allocator<char>`.

Каждый диспетчер типа `T` (каждый `Allocator<T>`) должен иметь функцию-член с именем `allocate`, такую как `a.allocate(n)`, возвращающую указатель на блок памяти с объемом, достаточным для хранения массива *n* объектов типа `T`. (Этот указатель будет приобретаться с помощью ресурса памяти, включающего экземпляр диспетчера памяти.) Стандарт не указывает, должна ли функция-член `allocate` быть статической и должна ли она принимать точно один параметр (`n`) или может принимать какие-то дополнительные параметры со значениями по умолчанию. Поэтому оба следующих класса типов в этом отношении являются допустимыми моделями `Allocator<int>`:

```
struct int_allocator_2014 {
  int *allocate(size_t n, const void *hint = nullptr);
};

struct int_allocator_2017 {
  int *allocate(size_t n);
};
```

Класс, обозначенный как `int_allocator_2017`, представляет, очевидно, *более простую* модель `Allocator<int>`, но `int_allocator_2014` тоже реализует правильную модель, и в обоих случаях выражение `a.allocate(n)` будет приниматься компилятором; и это все, что нам нужно, когда мы говорим об *обобщенном программировании*.

В классическом полиморфизме, напротив, мы определяем фиксированную сигнатуру для каждого метода базового класса, и производным классам не позволяется отклоняться от нее:

```
struct classical_base {
  virtual int *allocate(size_t n) = 0;
};

struct classical_derived : public classical_base {
  int *allocate(size_t n) override;
};
```

В производном классе `classical_derived` не получится добавить дополнительные параметры в сигнатуру метода `allocate`; получится изменить тип возвращаемого значения; получится сделать метод статическим. В классическом полиморфизме интерфейс «отливается в граните», в отличие от обобщенного программирования.

Поскольку «отлитый в граните» классический интерфейс легче описать, чем более широкий концептуальный, мы начнем наше знакомство с библиотекой диспетчеров памяти с совершенно нового, появившегося в C++17, классически полиморфного `memory_resource`.

Определение кучи с помощью memory_resource

Напомню, что на платформах с ограниченными ресурсами мы не можем использовать «кучу» (например, посредством `new` и `delete`), потому что среда выполнения такой платформы может не поддерживать распределение из динамической памяти. Но мы можем создать свою маленькую кучу – не «кучу», а всего лишь «кучку» – и смоделировать распределение динамической памяти, написав пару функций, выделяющих и освобождающих зарезервированные блоки памяти из большого статического массива типа `char`, например так:

```
static char big_buffer[10000];
static size_t index = 0;

void *allocate(size_t bytes) {
  if (bytes > sizeof big_buffer - index) {
    throw std::bad_alloc();
  }
  index += bytes;
  return &big_buffer[index - bytes];
}

void deallocate(void *p, size_t bytes) {
  // сбросить
}
```

Чтобы сохранить код максимально простым, я определил `deallocate` как пустую функцию. Эта маленькая куча позволит вызывающему коду выделить до 10 000 байт памяти, после чего начнет возбуждать исключения `bad_allocfrom`.

Приложив чуть больше усилий, можно позволить вызывающему коду не только выделять, но и освобождать память бесконечное число раз, при условии, что общий объем выделенной памяти не превысит 10 000 байт и вызывающий код неуклонно следует правилу «последний выделенный освобождается первым»:

```
void deallocate(void *p, size_t bytes) {
  if ((char*)p + bytes == &big_buffer[index]) {
    // ага! индекс можно открутить назад!
    index -= bytes;
  } else {
    // сбросить
  }
}
```

Самым существенным моментом здесь является наличие у нашей кучи некоторого состояния (в данном случае `big_buffer` и `index`) и пары функций, управляющих этим состоянием. Мы уже видели два возможных варианта реализации освобождения – есть и другие возможности, с дополнительным общим состоянием, не настолько подверженные «утечкам», – но интерфейс, сигнатуры функций `allocate` и `deallocate`, остается постоянным. Это говорит о том, что состояние и функции доступа можно завернуть в объект C++; а широкое разнообразие возможных реализаций плюс постоянство сигнатур функций предполагает, что мы можем использовать некоторую форму классического полиморфизма.

Модель диспетчера памяти в C++17 построена именно так. Стандартная библиотека предоставляет определение классически полиморфного базового класса, `std::pmr::memory_resource`, а мы на его основе реализуем свою маленькую кучу в производном классе. (На практике можно использовать один из производных классов, имеющихся в стандартной библиотеке, но давайте сначала закончим наш маленький пример и только потом поговорим об этом.) Базовый класс `std::pmr::memory_resource` определен в стандартном заголовке `<memory_resource>`:

```
class memory_resource {
  virtual void *do_allocate(size_t bytes, size_t align) = 0;
  virtual void do_deallocate(void *p, size_t bytes, size_t align) = 0;
  virtual bool do_is_equal(const memory_resource& rhs) const = 0;
public:
  void *allocate(size_t bytes, size_t align) {
    return do_allocate(bytes, align);
  }

  void deallocate(void *p, size_t bytes, size_t align) {
    return do_deallocate(p, bytes, align);
  }

  bool is_equal(const memory_resource& rhs) const {
    return do_is_equal(rhs);
  }
};
```

Обратите внимание на интересную прослойку между общедоступным интерфейсом класса и виртуальной реализацией. Обычно, следуя по пути класси-

ческого полиморфизма, у нас есть только один наборов методов, одновременно общедоступных и виртуальных; но в данной ситуации мы имеем общедоступный, невиртуальный интерфейс, который вызывает приватные виртуальные методы. Такое отделение интерфейса от реализации имеет несколько неявных преимуществ. Например, он не позволяет в дочерних классах выполнять вызовы `this->SomeBaseClass::allocate()` с использованием синтаксиса «прямого вызова виртуальных методов невиртуальным способом», но, честно говоря, главным преимуществом для нас является тот факт, что, определяя производный класс, мы вообще не можем использовать ключевое слово `public`. Так как мы определяем только *реализацию*, но не интерфейс, весь написанный нами код может быть приватным. Вот тривиальный пример маленькой кучи, подверженной утечкам:

```
class example_resource : public std::pmr::memory_resource {
  alignas(std::max_align_t) char big_buffer[10000];
  size_t index = 0;
  void *do_allocate(size_t bytes, size_t align) override {
    if (align > alignof(std::max_align_t) ||
        (-index % align) > sizeof big_buffer - index ||
        bytes > sizeof big_buffer - index - (-index % align))
    {
        throw std::bad_alloc();
    }
    index += (-index % align) + bytes;
    return &big_buffer[index - bytes];
  }

  void do_deallocate(void *, size_t, size_t) override {
    // сбросить
  }

  bool do_is_equal(const memory_resource& rhs) const override {
    return this == &rhs;
  }
};
```

Обратите внимание, что стандартная функция `std::pmr::memory_resource::allocate` принимает не только размер в байтах, но также шаг выравнивания. Мы должны гарантировать, что любой указатель, возвращаемый из `do_allocate`, имеет соответствующее выравнивание; например, если вызывающий код планирует хранить в памяти число `int`, мы выделяем необходимый объем, при этом он может потребовать четырехбайтное выравнивание.

Последнее, что следует отметить в нашем производном классе `example_resource`, – он представляет фактические ресурсы, управляемые нашей «кучкой»; то есть он фактически содержит, владеет и управляет буфером `big_buffer`, из которого выделяет память. Для любого данного буфера `big_buffer` в нашей программе будет существовать только один объект `example_resource`, управляющий этим буфером. Как уже говорилось выше: объекты типа `example_re-`

source являются «ресурсами памяти», и, следовательно, они *не* предназначены для копирования или перемещения; они поддерживают классическую объектно-ориентированную модель и не поддерживают семантику значения.

В стандартной библиотеке есть несколько разновидностей ресурсов памяти и все они являются производными от `std::pmr::memory_resource`. Давайте познакомимся с некоторыми из них.

Использование стандартных ресурсов памяти

Ресурсы памяти в стандартной библиотеке делятся на две разновидности. Некоторые являются настоящими классами типов, из которых можно создавать экземпляры; а некоторые – «анонимными» классами, доступными только через функции синглтона. Догадаться, к какой разновидности относится тот или иной ресурс, можно, подумав о том, могут ли быть «разными» два экземпляра типа или же тип всегда представлен единственным экземпляром.

Простейший ресурс памяти в заголовке `<memory_resource>` является «анонимным» синглтоном, доступным через `std::pmr::null_memory_resource()`. Определение этой функции выглядит примерно так:

```
class UNKNOWN : public std::pmr::memory_resource {
  void *do_allocate(size_t, size_t) override {
    throw std::bad_alloc();
  }

  void do_deallocate(void *, size_t, size_t) override {}
  bool do_is_equal(const memory_resource& rhs) const override {
    return this == &rhs;
  }
};

std::pmr::memory_resource *null_memory_resource() noexcept {
  static UNKNOWN singleton;
  return &singleton;
}
```

Обратите внимание, что функция возвращает указатель на экземпляр синглтона. Как правило, объекты `std::pmr::memory_resource` управляются с помощью указателей, потому что сами объекты `memory_resource` не могут перемещаться.

`null_memory_resource` кажется довольно бесполезным; все, что он делает, – возбуждает исключение, когда кто-то пытается распределить память с его помощью. Однако он может пригодиться, когда вы начнете использовать более сложные ресурсы памяти, к которым мы вскоре подойдем.

Следующий, более сложный ресурс памяти – синглтон, доступный через `std::pmr::new_delete_resource()`; для выделения и освобождения памяти он использует `::operator new` и `::operator delete`.

Теперь поговорим об именованных классах типов. Они определяют ресурсы, когда имеет смысл в одной программе создать несколько ресурсов одного типа. Возьмем для примера класс `std::pmr::monotonic_buffer_resource`. Этот ресурс почти ничем не отличается от нашего `example_resource`, кроме двух аспектов: он хранит не большой буфер в виде члена данных (в стиле `std::array`), а указатель на большой буфер, выделенный где-то еще (в стиле `std::vector`). И когда заканчивается первый большой буфер, вместо возбуждения исключения `bad_alloc` он пытается выделить второй буфер и начинает распределять блоки из этого буфера, пока не исчерпает его; после чего выделяется третий буфер... и т. д., пока не столкнется с невозможностью выделить дополнительный буфер. Как и наш `example_resource`, он не освобождает выделенную память, пока сам объект ресурса не будет уничтожен. Этот ресурс имеет один полезный предохранительный клапан: если вызвать метод `a.release()`, ресурс `monotonic_buffer_resource` освободит все буферы, удерживаемые им в данный момент, подобно методу `clear()` вектора.

Создавая ресурс типа `std::pmr::monotonic_buffer_resource`, нужно ответить на два вопроса: где находится первый буфер, и куда обращаться, когда этот буфер будет исчерпан? Ответ на первый вопрос определяется парой аргументов `void*` и `size_t`, которые описывают первый буфер (может быть `nullptr`); и ответ на второй вопрос определяется аргументом `std::pmr::memory_resource*`, который указывает на ресурс «выше по течению». Во втором аргументе вполне можно передать `std::pmr::new_delete_resource()`, чтобы выделять новые буферы с помощью `::operator new`. Также в нем можно передать `std::pmr::null_memory_resource()`, чтобы установить верхний предел на использование памяти данным конкретным ресурсом. Вот пример использования последнего:

```
alignas(16) char big_buffer[10000];

std::pmr::monotonic_buffer_resource a(
  big_buffer, sizeof big_buffer,
  std::pmr::null_memory_resource()
);

void *p1 = a.allocate(100);
assert(p1 == big_buffer + 0);

void *p2 = a.allocate(100, 16); // выравнивание
assert(p1 == big_buffer + 112);

// Теперь очистить все распределенные блоки и начать сначала.
a.release();
void *p3 = a.allocate(100);
assert(p3 == big_buffer + 0);

// Когда буфер исчерпается, произойдет обращение к вышестоящему ресурсу
// за получением следующего буфера... и не получит ничего.
```

```
try {
  a.allocate(9901);
} catch (const std::bad_alloc&) {
  puts("The null_memory_resource did its job!");
}
```

Если вы забыли, какой вышестоящий ресурс использует конкретный `monotonic_buffer_resource`, всегда можно вызвать `a.upstream_resource()`; этот метод вернет указатель на вышестоящий ресурс, переданный конструктору.

Выделение из ресурса пулов

Последняя разновидность ресурсов памяти из предоставляемых стандартной библиотекой C++17 – так называемый «ресурс пулов». Ресурс пулов управляет не просто одним большим буфером, как `example_resource`; и даже не непрерывной цепочкой буферов, как `monotonic_buffer_resource`. Он управляет множествами «блоков» разных размеров. Все блоки заданного размера хранятся вместе в одном «пуле», что дает нам возможность говорить: «пул блоков с размером 4», «пул блоков с размером 16» и т. д. Когда поступает запрос на выделение блока памяти размером *k*, ресурс пулов просматривает пул блоков с размером *k*, извлекает один и возвращает вызывающему коду. Если пул блоков с размером *k* пуст, тогда ресурс пулов пытается выделить больше блоков из вышестоящего ресурса. Кроме того, если поступает запрос на выделение блока с очень большим размером, для которого нет собственного пула, тогда ресурсу пулов разрешается передать запрос непосредственно вышестоящему ресурсу.

Ресурсы пулов имеют две разновидности: *синхронизированные* и *несинхронизированные*, то есть безопасные и небезопасные в многопоточной среде. Если вы собираетесь обращаться к ресурсу пулов одновременно из двух разных потоков, тогда вам следует использовать `std::pmr::synchronized_pool_resource`, а если вы точно никогда не будете делать этого и вам нужна максимальная скорость, используйте `std::pmr::unsynchronized_pool_resource`. (Кстати, `std::pmr::monotonic_buffer_resource` всегда безопасен в многопоточной среде; `new_delete_resource()` тоже безопасен, потому что действует только посредством операторов `new` и `delete`.)

Конструируя ресурс типа `std::pmr::synchronized_pool_resource`, вы должны определить три параметра: размеры блоков в его пулах, сколько блоков он должен объединить в «фрагмент», когда собирается запросить больше блоков у вышестоящего ресурса и сам вышестоящий ресурс. К сожалению, стандартный интерфейс оставляет желать лучшего, причем настолько, что я положа руку на сердце советую, если эти параметры действительно имеют для вас значение, реализовать свой ресурс, производный от `memory_resource`, и вообще не касаться версии из стандартной библиотеки. Синтаксис выражения этих параметров также не блещет удобством:

```
std::pmr::pool_options options;
options.max_blocks_per_chunk = 100;
```

```
options.largest_required_pool_block = 256;

std::pmr::synchronized_pool_resource a(
  options,
  std::pmr::new_delete_resource()
);
```

Обратите внимание, что нет никакого способа точно определить желаемые размеры блоков; эти хлопоты перекладываются на создателя реализации `synchronized_pool_resource`. Если вам повезет, он выберет размеры, соответствующие вашим требованиям; но лично я не стал бы полагаться на это предположение. Обратите также внимание на отсутствие возможности использовать разные вышестоящие ресурсы для блоков разных размеров, а также отдельный вышестоящий «резервный» ресурс, который использовался бы для обработки запросов на выделение блоков необычного размера.

Проще говоря, в обозримом будущем я бы воздержался от использования встроенных классов, производных от `pool_resource`. Но сама идея создания своих классов на основе `memory_resource` выглядит очень привлекательно. Если вас заинтересовала тема выделения памяти и управления своими маленькими кучками, я советую адаптировать `memory_resource` под свои нужды.

До сих пор мы говорили только о разных стратегиях распределения, «персонифицированных» разными классами, производными от `memory_resource`. Но нам еще нужно рассмотреть, как связать `memory_resource` с алгоритмами и контейнерами из стандартной библиотеки шаблонов. И для этого нам придется покинуть мир классического полиморфизма `memory_resource` и вернуться обратно в мир семантики значения C++03 STL.

500-головый стандартный диспетчер памяти

Модель стандартного диспетчера памяти должна была бы показаться удивительной в 2011. Далее мы посмотрим, как с помощью единственного типа C++ можно добиться всего из перечисленного ниже.

- Указать ресурс памяти, используемый для распределения памяти.
- Снабжать каждый выделенный указатель некоторыми метаданными, которые будут сопровождать его в течение всей жизни, до самого освобождения.
- Связать объект контейнера с конкретным ресурсом памяти и гарантировать «нерушимость» этой связи – этот объект контейнера всегда будет использовать указанную кучу для получения памяти.
- Связать *значение* контейнера с конкретным ресурсом памяти, чтобы контейнер мог эффективно перемещаться с использованием семантики значения и не забывал при этом, как освобождать свое содержимое.
- Возможность выбирать между двумя взаимоисключающими поведениями выше.

- Задавать стратегию выделения памяти на всех уровнях многоуровневого контейнера, такого как вектор векторов.
- Переопределять понятие «конструирования» содержимого контейнера, чтобы, например, для `vector<int>::resize` можно было определить значение по умолчанию для инициализации вместо инициализации нулями.

Это просто *безумное* количество голов для одного класса типов – серьезное нарушение принципа единственной ответственности. Но именно это делает модель стандартного диспетчера памяти; поэтому попробуем разобраться во всех этих особенностях.

Как вы наверняка помните, «стандартный диспетчер памяти» – это любой класс, удовлетворяющий понятию `Allocator<T>` для некоторого типа `T`. В стандартной библиотеке имеется три типа стандартных диспетчеров памяти: `std::allocator<T>`, `std::pmr::polymorphic_allocator<T>` и `std::scoped_allocator_adaptor<A...>`.

Сначала рассмотрим `std::allocator<T>`:

```
template<class T>
struct allocator {
  using value_type = T;

  T *allocate(size_t n) {
    return static_cast<T *>(::operator new(n * sizeof (T)));
  }

  void deallocate(T *p, size_t) {
    ::operator delete(static_cast<void *>(p));
  }

  // ПРИМЕЧАНИЕ 1
  template<class U>
  explicit allocator(const allocator<U>&) noexcept {}

  // ПРИМЕЧАНИЕ 2
  allocator() = default;
  allocator(const allocator&) = default;
};
```

`std::allocator<T>` имеет функции-члены `allocate` и `deallocate`, требуемые идеей `Allocator<T>`. Напомню, что сейчас мы находимся в концептуальном мире обобщенного программирования! Классически полиморфный `memory_resource` также имеет функции-члены с именами `allocate` и `deallocate`, но они всегда возвращают `void*`, а не `T*`. (Кроме того, `memory_resource::allocate()` принимает два аргумента – `bytes` и `align`, – тогда как `allocator<T>::allocate()` принимает только один аргумент. Первая причина этого в том, что появление `allocator<T>` предшествовало появлению понимания важности выравнивания; напомню, что оператор `sizeof` был уна-

следован из C в 1980-х, а оператор `alignof` появился только в C++11. Вторая причина в том, что в контексте `std::allocator<T>` память выделяется для объектов типа `T`, поэтому выравнивание обязательно должно быть `alignof(T)`. `std::allocator<T>` не использует эту информацию, потому что появился раньше `alignof`, но, в принципе, это возможно, и именно поэтому `Allocator<T>` требует только сигнатуру `a.allocate(n)`, а не `a.allocate(n, align)`.)

Конструктор, отмеченный комментарием `ПРИМЕЧАНИЕ 1`, играет важную роль; каждый диспетчер памяти должен иметь шаблонный конструктор. Конструкторы, следующие за комментарием `ПРИМЕЧАНИЕ 2`, не играют существенной роли; единственная причина, по которой я добавил их, – если не определить их явно, они будут удалены из-за присутствия пользовательского конструктора (с комментарием `ПРИМЕЧАНИЕ 1`).

Идея любого стандартного диспетчера памяти заключается в возможности подключить его через последний шаблонный параметр типа к любому стандартному контейнеру (глава 4 «Зоопарк контейнеров»), после чего контейнер будет использовать его вместо обычного механизма для распределения памяти, независимо от причины. Рассмотрим пример:

```
template<class T>
struct helloworld {
  using value_type = T;

  T *allocate(size_t n) {
    printf("hello world %zu\n", n);
    return static_cast<T *>(::operator new(n * sizeof (T)));
  }
  void deallocate(T *p, size_t) {
    ::operator delete(static_cast<void *>(p));
  }
};

void test() {
  std::vector<int, helloworld<int>> v;
  v.push_back(42); // выведет "hello world 1"
  v.push_back(42); // выведет "hello world 2"
  v.push_back(42); // выведет "hello world 4"
}
```

Здесь наш класс `helloworld<int>` моделирует `Allocator<int>`, но мы опустили шаблонный конструктор. Это нормально, если он будет использоваться только в паре с `vector`, потому что `vector` выделяет только массивы с типом его элементов. Но посмотрите, что получится, если изменить функцию `test`, как показано ниже:

```
void test() {
  std::list<int, helloworld<int>> v;
  v.push_back(42);
}
```

Для libc++ этот код породит несколько десятков строк с сообщениями об ошибках, которые сводятся к единственному: "no known conversion from helloworld<int> to helloworld<std::__1::__list_node<int, void *>>" (неизвестное преобразование из helloworld<int> в helloworld<std::__1::__list_node<int, void *>>). Как было показано на рис. 4.5, в главе 4 «Зоопарк контейнеров» std::list<T> хранит свои элементы в узлах, размер которых больше размера типа T. Поэтому std::list<T> пытается выделить память не для объектов типа T, а для объектов типа __list_node. А для этого ему нужен диспетчер памяти, моделирующий идею Allocator<__list_node>, а не Allocator<int>.

Внутренне конструктор std::list<int> принимает наш helloworld<int> и пытается «перепривязать» его, чтобы выделить память для объектов __list_node, а не int. Это достигается посредством идиомы traits class--a, с которой мы впервые столкнулись в главе 2 «Итераторы и диапазоны»:

```
using AllocOfInt = helloworld<int>;

using AllocOfChar =
  std::allocator_traits<AllocOfInt>::rebind_alloc<char>;

// Теперь alloc_of_char - это helloworld<char>
```

Стандартный шаблонный класс std::allocator_traits<A> заключает большой объем информации о типе диспетчера A, поэтому ее легко получить. Например, std::allocator_traits<A>::value_type – это псевдоним типа T, память для которого выделяется диспетчером A; а std::allocator_traits<A>::pointer – это псевдоним соответствующего типа указателя (обычно T*).

Вложенный шаблон псевдонима std::allocator_traits<A>::rebind_alloc<U> – это способ «преобразовать» тип диспетчера из T в U. Этот прием метапрограммирования используется, чтобы вскрыть тип A и увидеть: во-первых, имеет ли A вложенный шаблон псевдонима A::rebind<U>::other (что бывает редко), и, во-вторых, можно ли выразить тип A в форме Foo<Bar,Baz...> (где Baz... – некоторый список типов, который может быть пустым). Если A можно выразить таким способом, тогда std::allocator_traits<A>::rebind_alloc<U> будет служить синонимом для Foo<U,Baz...>. С философской точки зрения в этом мало смысла; но на практике этот прием работает для всех типов диспетчеров, которые вам могут встретиться. В частности, он работает для helloworld<int> – что объясняет, почему нам не пришлось возиться с предоставлением вложенного псевдонима rebind<U>::other в нашем классе helloworld. Предоставляя разумное поведение по умолчанию, шаблон std::allocator_traits избавил нас от необходимости писать массу типового кода. В этом заключается причина существования std::allocator_traits.

Возможно, вам интересно, почему std::allocator_traits<Foo<Bar,Baz...>>::value_type не работает по умолчанию для Bar. Признаться честно,

я сам не знаю. Я не вижу в этом большой проблемы, но стандартная библиотека не делает этого. Поэтому каждый тип диспетчера, который вы пишете (не забывайте, что сейчас речь идет о классах, моделирующих `Allocator<T>`, а *не* о классах, производных от `memory_resource`), должен предоставлять вложенное определение типа `value_type`, который является псевдонимом для `T`.

Однако после определения вложенного типа для `value_type` можно быть уверенным, что `std::allocator_traits` выведет правильные определения для его вложенного типа `pointer` (то есть `T*`), и `const_pointer` (`const T*`), и `void_pointer` (`void*`), и т. д. Следившие за предыдущим обсуждением `rebind_alloc` могли бы предположить, что «преобразование» типа указателя, такого как `T*`, в `void*`, не более сложно или просто, чем «преобразование» типа диспетчера `Foo<T>` в `Foo<void>`, и будут правы! Значения этих псевдонимов типов, связанных с указателями, определяются посредством второго стандартного класса трейта `std::pointer_traits<P>`:

```
using PtrToInt = int*;

using PtrToChar =
  std::pointer_traits<PtrToInt>::rebind<char>;

// Теперь PtrToChar имеет тип char*

using PtrToConstVoid =
  std::pointer_traits<PtrToInt>::rebind<const void>;

// Теперь PtrToConstVoid имеет тип const void*
```

Важность этого класса трейта становится особенно очевидной, когда речь заходит о следующей области ответственности `Allocator<T>`: «снабдить каждый выделенный указатель некоторыми метаданными, которые будут сопровождать его на протяжении всего жизненного цикла».

Метаданные, сопровождающие причудливые указатели

Рассмотрим следующую высокоуровневую архитектуру ресурса памяти, напоминающего `std::pmr::monotonic_buffer_resource`.

- Хранит список фрагментов памяти, полученных от системы. Для каждого фрагмента хранится также `index` – количество байтов, выделенных от начала этого фрагмента, и счетчик `freed` – количество байтов, возвращенных в этот фрагмент.
- Когда какой-то код вызывает `allocate(n)`, увеличивается `index` фрагмента на запрошенное количество байтов, если возможно, или у вышестоящего ресурса запрашивается новый фрагмент.
- Когда какой-то код вызывает `deallocate(p, n)`, выясняется, какому из фрагментов принадлежит адрес `p`, и увеличивается его счетчик

`freed += n`. Если `freed == index`, тогда считается, что весь фрагмент свободен, поэтому выполняется `freed = index = 0`.

Это описание довольно легко преобразуется в программный код. Единственная проблема: как в `deallocate(p, n)` определить, какому из фрагментов принадлежит адрес `p`?

Это легко сделать, если записать идентичность фрагмента в сам «указатель»:

```
template<class T>
class ChunkyPtr {
  T *m_ptr = nullptr;
  Chunk *m_chunk = nullptr;
public:
  explicit ChunkyPtr(T *p, Chunk *ch) :
  m_ptr(p), m_chunk(ch) {}

  T& operator *() const {
    return *m_ptr;
  }
  explicit operator T *() const {
    return m_ptr;
  }

  // ... и т. д. ...

  // ... плюс следующий дополнительный метод доступа:
  auto chunk() const {
    return m_chunk;
  }
};
```

После этого нам останется только вызвать `p.chunk()` в функции `deallocate(p, n)`. Но для этого придется изменить сигнатуры функций `allocate(n)` и `deallocate(p, n)`, чтобы `deallocate` принимала `ChunkyPtr<T>` вместо `T*`, а `allocate` возвращала `ChunkyPtr<T>` вместо `T*`.

К счастью, стандартная библиотека C++ дает такую возможность! Мы должны лишь определить свой тип, моделирующий `Allocator<T>`, и добавить в него определение типа `pointer` как `ChunkyPtr<T>`:

```
template<class T>
struct ChunkyAllocator {
  using value_type = T;
  using pointer = ChunkyPtr<T>;

  ChunkyAllocator(ChunkyMemoryResource *mr) :
    m_resource(mr) {}

  template<class U>
  ChunkyAllocator(const ChunkyAllocator<U>& rhs) :
```

```
    m_resource(rhs.m_resource) {}

  pointer allocate(size_t n) {
    return m_resource->allocate(
      n * sizeof(T), alignof(T));
  }

  void deallocate(pointer p, size_t n) {
    m_resource->deallocate(
      p, n * sizeof(T), alignof(T));
  }
private:
  ChunkyMemoryResource *m_resource;

  template<class U>
  friend struct ChunkyAllocator;
};
```

Классы трейтов `std::allocator_traits` и `std::pointer_traits` позаботятся о выводе других определений типов, таких как `void_pointer`, который благодаря волшебству `pointer_traits::rebind` превратится в псевдоним для `ChunkyPtr<void>`.

Я не стал приводить здесь реализацию функций `allocate` и `deallocate`, потому что они зависят от интерфейса `ChunkyMemoryResource`. Мы могли бы реализовать `ChunkyMemoryResource` примерно так:

```
class Chunk {
  char buffer[10000];
  int index = 0;
  int freed = 0;
public:
  bool can_allocate(size_t bytes) {
    return (sizeof buffer - index) >= bytes;
  }
  auto allocate(size_t bytes) {
    index += bytes;
    void *p = &buffer[index - bytes];
    return ChunkyPtr<void>(p, this);
  }
  void deallocate(void *, size_t bytes) {
    freed += bytes;
    if (freed == index) {
      index = freed = 0;
    }
  }
};

class ChunkyMemoryResource {
  std::list<Chunk> m_chunks;
public:
  ChunkyPtr<void> allocate(size_t bytes, size_t align) {
```

```
    assert(align <= alignof(std::max_align_t));
    bytes += -bytes % alignof(std::max_align_t);
    assert(bytes <= 10000);

    for (auto&& ch : m_chunks) {
      if (ch.can_allocate(bytes)) {
        return ch.allocate(bytes);
      }
    }
    return m_chunks.emplace_back().allocate(bytes);
  }
  void deallocate(ChunkyPtr<void> p, size_t bytes, size_t) {
    bytes += -bytes % alignof(std::max_align_t);
    p.chunk()->deallocate(static_cast<void*>(p), bytes);
  }
};
```

Теперь класс `ChunkyMemoryResource` можно использовать для выделения памяти стандартными контейнерами, поддерживающими работу с диспетчерами памяти:

```
ChunkyMemoryResource mr;
std::vector<int, ChunkyAllocator<int>> v{&mr};
v.push_back(42);
// Вся память для внутреннего массива в векторе v
// будет получена из блоков, которыми владеет "mr".
```

Я выбрал этот пример, стремясь сделать его как можно проще и понятнее; но совсем забыл о деталях самого типа `ChunkyPtr<T>`. Если вы попытаетесь воспроизвести этот код у себя, то обнаружите, что должны сопроводить `ChunkyPtr` множеством перегруженных операторов, таких как `==`, `!=`, `<`, `++`, `--` и `-`; и вам также потребуется определить специализацию для `ChunkyPtr<void>`, с отсутствующим перегруженным `operator*`. Большинство этих деталей знакомо вам по главе 2 «Итераторы и диапазоны», где мы реализовали пример своего типа итератора. Фактически каждый тип «причудливого указателя» должен поддерживать возможность использования в роли итератора с произвольным доступом, а это означает, что вы должны добавить пять вложенных определений типов, перечисленных в конце главы 2: `iterator_category`, `difference_type`, `value_type`, `pointer` и `reference`.

Наконец, если вы пожелаете использовать какие-то конкретные контейнеры, такие как `std::list` и `std::map`, вам придется реализовать статическую функцию-член со странным именем `pointer_to(r)`:

```
static ChunkyPtr<T> pointer_to(T &r) noexcept {
  return ChunkyPtr<T>(&r, nullptr);
}
```

Это связано с тем, что – как рассказывалось в главе 4 «Зоопарк контейнеров» – некоторые контейнеры, такие как `std::list`, хранят свои данные в уз-

лах, указатели `prev` и `next` в которых должны иметь возможность указывать *либо* на узел, выделенный в памяти, *либо* на узел, находящийся внутри члена данных самого объекта `std::list`. Есть два очевидных способа добиться этого: хранить каждый указатель `next` в виде некоторого размеченного объединения (tagged union) с причудливым и обычным указателями (возможно, `std::variant`, как описывалось в главе 5 «Словарные типы») или найти способ закодировать простой указатель *в виде* причудливого указателя. В стандартной библиотеке выбран второй подход. То есть всякий раз, реализуя причудливый указатель, вы должны не только сделать то, что требует диспетчер памяти и итератор с произвольным доступом, но также предусмотреть способ представления любого произвольного указателя в адресном пространстве программы – по крайней мере, если хотите использовать диспетчер с контейнерами, основанными на использовании узлов, такими как `std::list`.

Но, пройдя через все эти сложности, вы обнаружите, что (на момент издания этой книги) ни libc++, ни libstdc++ не поддерживают причудливых указателей в контейнерах, сложнее чем `std::vector`. Они обладают достаточной поддержкой только для работы с единственным типом причудливых указателей – `boost::interprocess::offset_ptr<T>`, – который не сопровождается никакими метаданными. Кроме того, стандарт продолжает развиваться; тип `std::pmr::memory_resource` только-только появился в C++17 и на момент написания этих строк все еще оставался нереализованным в libc++ и libstdc++.

Возможно, вы заметили также отсутствие любых стандартных базовых классов ресурсов памяти, использующих причудливые указатели. К счастью, вы легко сможете написать их сами:

```
namespace my {

  template<class VoidPtr>
  class fancy_memory_resource {
  public:
    VoidPtr allocate(size_t bytes,
      size_t align = alignof(std::max_align_t)) {
      return do_allocate(bytes, align);
    }
    void deallocate(VoidPtr p, size_t bytes,
      size_t align = alignof(std::max_align_t)) {
      return do_deallocate(p, bytes, align);
    }
    bool is_equal(const fancy_memory_resource& rhs) const noexcept {
      return do_is_equal(rhs);
    }
    virtual ~fancy_memory_resource() = default;
  private:
    virtual VoidPtr do_allocate(size_t bytes, size_t align) = 0;
    virtual void do_deallocate(VoidPtr p, size_t bytes, size_t align) = 0;
    virtual bool do_is_equal(const fancy_memory_resource& rhs)
      const noexcept = 0;
```

```
  };

  using memory_resource = fancy_memory_resource<void*>;

} // namespace my
```

В стандартной библиотеке отсутствуют диспетчеры памяти, использующие причудливые указатели; все библиотечные диспетчеры используют обычные указатели.

Прикрепление контейнера к единственному ресурсу памяти

Следующая голова модели стандартного диспетчера памяти – следующая особенность, которой управляет `std::allocator_traits` – возможность связать конкретные объекты контейнеров с определенными кучами. Эта особенность была описана выше следующими тремя пунктами:

- связать объект контейнера с конкретным ресурсом памяти и гарантировать «нерушимость» этой связи – этот объект контейнера всегда будет использовать указанную кучу для получения памяти;
- связать *значение* контейнера с конкретным ресурсом памяти, чтобы контейнер мог эффективно перемещаться с использованием семантики значения и не забывал при этом, как освобождать свое содержимое;
- возможность выбирать между двумя взаимоисключающими поведениями выше.

Рассмотрим пример, в котором в качестве ресурса используем `std::pmr::monotonic_buffer_resource` и свой класс диспетчера памяти. (Просто чтобы вы могли быть уверены, что ничего не пропустили: в действительности мы пока не рассмотрели ни одного типа диспетчера из стандартной библиотеки, кроме `std::allocator<T>` – простейшего диспетчера без состояния, который выделяет и освобождает память из глобальной кучи посредством операторов `new` и `delete`.)

```
template<class T>
struct WidgetAlloc {
  std::pmr::memory_resource *mr;

  using value_type = T;

  WidgetAlloc(std::pmr::memory_resource *mr) : mr(mr) {}

  template<class U>
  WidgetAlloc(const WidgetAlloc<U>& rhs) : mr(rhs.mr) {}

  T *allocate(size_t n) {
```

```
    return (T *)mr->allocate(n * sizeof(T), alignof(T));
  }

  void deallocate(void *p, size_t n) {
    mr->deallocate(p, n * sizeof(T), alignof(T));
  }
};

class Widget {
  char buffer[10000];
  std::pmr::monotonic_buffer_resource mr {buffer, sizeof buffer};
  std::vector<int, WidgetAlloc<int>> v {&mr};
  std::list<int, WidgetAlloc<int>> lst {&mr};
public:
  static void swap_elems(Widget& a, Widget& b) {
    std::swap(a.v, b.v);
  }
};
```


Это наш класс `Widget` – классический объектно-ориентированный тип. Предполагается, что весь свой жизненный цикл его экземпляр будет находиться в определенной области памяти. Поэтому, чтобы уменьшить фрагментацию кучи и улучшить локальность кеша, мы решили добавить большой буфер в каждый объект `Widget` и использовать этот буфер как для членов данных `v` и `lst`.

Теперь рассмотрим функцию `Widget::swap_elems(a, b)`. Она меняет местами члены данных `v` двух экземпляров, `Widget a` и `Widget b`. Как рассказывалось в главе 4 «Зоопарк контейнеров», `std::vector` – это чуть больше, чем указатель на массив, размещенный в динамической памяти, поэтому, чтобы поменять местами два экземпляра `std::vector`, *обычно* достаточно поменять их внутренние указатели, не перемещая самого содержимого векторов, благодаря чему операция `swap` для векторов получает сложность O(1) вместо O(*n*).

Кроме того, тип `vector` достаточно интеллектуальный, чтобы знать, что кроме указателей нужно также поменять местами диспетчеры памяти, чтобы информация о порядке освобождения памяти перемещалась вместе с указателем, который в конечном счете потребуется освободить.

Но в этом случае, если библиотека просто поменяет местами указатели и диспетчеры памяти, это станет катастрофой! Мы получим вектор `a.v`, чьим внутренним массивом теперь будет «владеть» `b.mr`, и наоборот. Если мы решим уничтожить `Widget b`, тогда при очередной попытке доступа к элементам `a.v` мы обратимся к освобожденной памяти. Но даже если мы никогда не будем обращаться к `a.v`, наша программа почти наверняка завершится аварийно, когда деструктор `a.v` попытается вызвать метод `deallocate` давно уничтоженного `b.mr`!

К счастью, стандартная библиотека хранит нас от такой судьбы. Одной из обязанностей контейнера с поддержкой выбора диспетчера памяти является *передача* этого диспетчера в операции присваивания копированием и перемещением, а также в операцию обмена местами. По исторически сложившимся

причинам это обрабатывается путем определений типов в шаблонном классе `allocator_traits`, но для правильного использования передачи диспетчера памяти достаточно знать следующее:

- будет ли передаваться диспетчер памяти или жестко прикрепляться к определенному контейнеру, зависит от свойств *типа диспетчера*. Если потребуется, чтобы один диспетчер «закреплялся», а другой передавался, вы *должны* сделать их разными типами;
- «закрепляемый» диспетчер прикрепляется к конкретному объекту контейнера (классическим объектно-ориентированным способом). Операции обмена указателями, которые для «незакрепляемых» типов диспетчеров могут иметь сложность O(1), в «закрепляемых» типах получают сложность O(*n*), потому что для переноса элементов из-под управления одного диспетчера под управление другого требуется вновь выделить память;
- «закрепляемые» диспетчеры имеют четко очерченный круг применения (как в примере с классом `Widget`), и последствия использования «незакрепляемых» диспетчеров иногда могут оказаться катастрофическими (и снова вспомните пример класса `Widget`). Поэтому `std::allocator_traits` по умолчанию предполагает, что тип диспетчера памяти является «закрепляемым», если не может сказать, что он *пустой* и потому определенно *не имеет состояния*. По умолчанию для *пустых* типов диспетчеров предполагается невозможность «закрепления»;
- как программист вы почти всегда будете следовать за умолчаниями: использовать диспетчеры без состояния там, где требуется их передача, и почти никогда не использовать диспетчеры с состоянием за рамками сценариев в стиле `Widget`.

Использование диспетчеров памяти стандартных типов

Давайте поговорим о типах диспетчеров памяти, имеющихся в стандартной библиотеке.

`std::allocator<T>` – тип диспетчера памяти по умолчанию; это значение по умолчанию для шаблонного параметра типа любого контейнера. Например, когда в своем коде вы объявляете вектор `std::vector<T>`, за кулисами компилятор определит его как тип `std::vector<T, std::allocator<T>>`. Как уже упоминалось выше в этой главе, `std::allocator<T>` – это пустой тип, не имеющий состояния; он использует глобальную кучу посредством операторов `new` и `delete`. Так как тип `std::allocator` не имеет состояния, `allocator_traits` предполагает (совершенно справедливо), что его нельзя прикрепить к контейнеру. Это означает, что такие операции, как `std::vector<T>::swap` и `std::vector<T>::operator=`, гарантированно будут выполняться очень эффективно, простой сменой ука-

зателей, потому что любой объект типа `std::vector<T, std::allocator<T>>` всегда знает, как освободить память, прежде выделенную другим экземпляром `std::vector<T, std::allocator<T>>`.

`std::pmr::polymorphic_allocator<T>` – новый тип, появившийся в C++17. Это непустой тип с состоянием; его единственный член данных – указатель на `std::pmr::memory_resource`. (Фактически он почти идентичен `WidgetAlloc` из нашего примера выше!) Два разных экземпляра `std::pmr::polymorphic_allocator<T>` не всегда могут оказаться взаимозаменяемыми, потому что их указатели могут указывать на совершенно разные ресурсы `memory_resource`; это значит, что объект типа `std::vector<T, std::pmr::polymorphic_allocator<T>>` может не знать, как освободить память, распределенную некоторым другим экземпляром `std::vector<T, std::pmr::polymorphic_allocator<T>>`. А это, в свою очередь, значит, что `std::pmr::polymorphic_allocator<T>` является «закрепляемым» типом диспетчера памяти, то есть операции, такие как `std::vector<T, std::pmr::polymorphic_allocator<T>>::operator=`, могут приводить к копированию больших объемов данных.

К слову сказать, довольно утомительно снова и снова писать полное имя типа `std::vector<T, std::pmr::polymorphic_allocator<T>>`. К счастью, производители реализаций стандартной библиотеки заметили это и добавили псевдонимы типа в пространство имен `std::pmr`:

```
namespace std::pmr {

  template<class T>
  using vector = std::vector<T,
    polymorphic_allocator<T>>;

  template<class K, class V, class Cmp = std::less<K>>
  using map = std::map<K, V, Cmp,
    polymorphic_allocator<typename std::map<K, V>::value_type>>;

  // ...

} // namespace std::pmr
```

Настройка ресурса памяти по умолчанию

Главное отличие стандартного `polymorphic_allocator` от `WidgetAlloc` в нашем примере – стандартный `polymorphic_allocator` конструируется по умолчанию. Возможно, это привлекательная черта диспетчера памяти; она означает, что первую строку (см. ниже) можно записать как вторую:

```
std::pmr::vector<int> v2({1, 2, 3}, std::pmr::new_delete_resource());
  // Определение конкретного ресурса памяти

std::pmr::vector<int> v1 = {1, 2, 3};
  // Использование ресурса памяти по умолчанию
```

С другой стороны, взглянув на вторую строку, вы можете спросить: «Где действительно размещается внутренний массив?» В конце концов, диспетчер памяти определяется в основном, чтобы точно знать, откуда будут взяты необходимые байты памяти! Вот почему при обычном способе конструирования стандартного `polymorphic_allocator` передается указатель на `memory_resource` – фактически предполагается, что эта идиома станет настолько общей, что преобразование из `std::pmr::memory_resource*` в `std::pmr::polymorphic_allocator` стало неявным. Но в `polymorphic_allocator` тоже есть конструктор по умолчанию (без аргументов). Создавая `polymorphic_allocator` по умолчанию, вы получаете дескриптор «ресурса памяти по умолчанию» – `new_delete_resource()`. Но вы можете изменить это поведение! Указатель на ресурс памяти по умолчанию хранится в глобальной атомарной (безопасной в многопоточной среде) переменной, управление содержимым которой осуществляется посредством функций `std::pmr::get_default_resource()` (возвращает указатель) и `std::pmr::set_default_resource()` (сохраняет новый указатель и возвращает прежний).

Если вы захотите полностью избежать выделения памяти из кучи посредством `new` и `delete`, тогда имеет смысл вызвать `std::pmr::set_default_resource(std::pmr::null_memory_resource())` в начале программы. Конечно, вы не сможете помешать другой части программы нарушить ваш замысел и также вызвать `set_default_resource`; а так как всеми потоками используется одна и та же глобальная переменная, можно столкнуться с очень странным поведением, если попытаться менять ресурс по умолчанию во время работы программы. Нельзя, например, сказать: «установить этот ресурс памяти по умолчанию для текущего потока». Кроме того, вызов `get_default_resource()` (например, из конструктора по умолчанию `polymorphic_allocator`) реализует атомарный доступ, из-за чего выполняется чуть медленнее. Поэтому лучше избегать конструктора по умолчанию `polymorphic_allocator` – всегда явно указывайте, какой ресурс памяти пытаетесь использовать. В случаях, когда требуется абсолютная надежность, можно даже подумать об использовании чего-то подобного `WidgetAlloc` из примера выше вместо `polymorphic_allocator`; *отсутствие* конструктора по умолчанию не позволит использовать `WidgetAlloc` неправильно.

Создание контейнера с поддержкой выбора диспетчера памяти

После знакомства с ресурсами памяти (кучами) и диспетчерами (дескрипторами куч) обратим взгляд на третью ногу треноги: классы контейнеров. Внутренне каждый контейнер с поддержкой выбора диспетчера памяти должен, по меньшей мере:

- хранить экземпляр диспетчера как член данных (то есть контейнер должен принимать шаблонный параметр с типом диспетчера, иначе он не будет знать, как много места зарезервировать для переменной-члена);

- иметь конструктор, принимающий аргумент с диспетчером памяти;
- фактически использовать диспетчер для выделения и освобождения памяти; полностью исключить любое использование `new` или `delete`;
- конструктор перемещения контейнера, оператор присваивания перемещением и функция `swap` должны передавать диспетчер согласно его `allocator_traits`.

Вот пример очень простого контейнера с поддержкой выбора диспетчера памяти – контейнера, который размещает в куче единственный объект. Это версия `std::unique_ptr<T>` из главы 6 «Умные указатели»:

```
template<class T, class A = std::allocator<T>>
class uniqueish {
  using Traits = std::allocator_traits<A>;
  using FancyPtr = typename Traits::pointer;

  A m_allocator;
  FancyPtr m_ptr = nullptr;

public:
  using allocator_type = A;

  uniqueish(A a = {}) : m_allocator(a) {
    this->emplace();
  }

  ~uniqueish() {
    clear();
  }

  T& value() { return *m_ptr; }
  const T& value() const { return *m_ptr; }

  template<class... Args>
  void emplace(Args&&... args) {
    clear();
    m_ptr = Traits::allocate(m_allocator, 1);
    try {
      T *raw_ptr = static_cast<T *>(m_ptr);
      Traits::construct(m_allocator, raw_ptr,
        std::forward<Args>(args)...
      );
    } catch (...) {
      Traits::deallocate(m_allocator, m_ptr, 1);
      throw;
    }
  }

  void clear() noexcept {
    if (m_ptr) {
```

```
      T *raw_ptr = static_cast<T *>(m_ptr);
      Traits::destroy(m_allocator, raw_ptr);
      Traits::deallocate(m_allocator, m_ptr, 1);
      m_ptr = nullptr;
    }
  }
};
```

Обратите внимание, что там, где `unique_ptr` использует `T*`, наш код использует `allocator_traits<A>::pointer`; а там, где `make_unique` использует `new` и `delete`, наш код использует пару вызовов `allocator_traits<A>::allocate/construct` и `allocator_traits<A>::destroy/deallocate`. Мы уже обсуждали назначение `allocate` и `deallocate` – они управляют выделением памяти из соответствующего ресурса. Но этот фрагмент памяти – всего лишь массив байтов; чтобы превратить его в объект, мы должны сконструировать экземпляр `T` по этому адресу. Мы могли бы использовать синтаксис «размещающей формы `new`» (placement-new); но в следующем разделе мы увидим, почему так важно вместо нее использовать `construct` и `destroy`.

Наконец, обратите внимание, что деструктор `uniqueish` проверяет наличие распределенной памяти, перед тем как освободить ее. Это важно, потому что позволяет нам иметь значение `uniqueish`, представляющее «пустой объект», – значение, которое можно сконструировать без выделения памяти и использовать как «перемещенное» представление для нашего типа.

Теперь реализуем операции перемещения для нашего типа. Нам хотелось бы, чтобы после перемещения объекта `uniqueish<T>` по прежнему адресу оказался «пустой объект». Кроме того, если оба объекта, слева и справа, используют один и тот же диспетчер памяти или если тип диспетчера нельзя закрепить за контейнером, тогда желательно было бы вообще избежать применения конструктора перемещения `T`, а передать владение выделенным указателем из объекта справа объекту слева:

```
uniqueish(uniqueish&& rhs) : m_allocator(rhs.m_allocator)
{
  m_ptr = std::exchange(rhs.m_ptr, nullptr);
}

uniqueish& operator=(uniqueish&& rhs)
{
  constexpr bool pocma =
    Traits::propagate_on_container_move_assignment::value;
  if constexpr (pocma) {
    // Мы можем принять новый диспетчер памяти, потому что
    // наш диспетчер не является "закрепляемым".
    this->clear(); // используется прежний диспетчер
    this->m_allocator = rhs.m_allocator;
    this->m_ptr = std::exchange(rhs.m_ptr, nullptr);
  } else if (m_allocator() == rhs.m_allocator()) {
    // Наш диспетчер закреплен за этим контейнером;
```

```
      // но поскольку он эквивалентен диспетчеру в rhs,
      // мы можем использовать память из rhs.
      this->clear();
      this->m_ptr = std::exchange(rhs.m_ptr, nullptr);
    } else {
      // Мы не должны передавать этот новый диспетчер
      // и поэтому не можем использовать его память.
      if (rhs.m_ptr) {
        this->emplace(std::move(rhs.value()));
        rhs.clear();
      } else {
        this->clear();
      }
    }
    return *this;
  }
```

Конструктор перемещения реализуется так же просто, как прежде. Единственное небольшое отличие – мы должны не забыть сконструировать наш `m_allocator`, копию диспетчера памяти в объекте справа.

Мы могли не копировать, а переместить диспетчер памяти с помощью `std::move`, но я не думаю, что для данного примера это имеет какое-то значение. Не забывайте, что диспетчер памяти – это лишь «дескриптор», указывающий на фактический ресурс памяти, и что в действительности многие типы диспетчеров, такие как `std::allocator<T>`, – «пустые». Копирование типа диспетчера практически всегда – относительно недорогая операция. Однако и использование `std::move` здесь не нанесло бы большого ущерба.

Оператор *присваивания перемещением*, напротив, очень сложен! В первую очередь нужно проверить «закрепляемость» типа диспетчера. «Незакрепляемость» обозначается значением `true` в `propagate_on_container_move_assignment::value`, или более кратко `pocma`. (Фактически стандарт утверждает, что значение `propagate_on_container_move_assignment` должно совпадать с `std::true_type`; и GNU libstdc++ будет твердо придерживаться этого требования. Поэтому имейте это в виду, когда будете определять свои типы диспетчеров памяти.) Для незакрепляемых типов диспетчеров при присваивании перемещением, эффективнее уничтожить текущее значение (если имеется) – с использованием прежнего диспетчера `m_allocator`, – а затем использовать объект справа с его диспетчером. Поскольку указатель перемещается с диспетчером, можно быть полностью уверенным, что потом мы сможем освободить память.

С другой стороны, если диспетчер памяти относится к «закрепляемому» типу, нельзя использовать диспетчер из объекта справа. Если текущий (закреп-

ленный) экземпляр диспетчера равен экземпляру диспетчера в объекте справа, тогда можно смело принять указатель из объекта справа, потому что мы уже знаем, как освободить память, выделенную данным конкретным экземпляром диспетчера.

Наконец, если мы не можем принять экземпляр диспетчера из объекта справа и наш текущий экземпляр диспетчера не равен экземпляру в объекте справа, значит мы не можем принять его, потому что позднее не сможем освободить этот указатель, и единственная возможность сделать это – использовать экземпляр диспетчера правого объекта прямо сейчас. В этом случае мы вынуждены распределить совершенно новый фрагмент памяти, используя свой экземпляр диспетчера, а затем скопировать данные из `rhs.value()` в свое значение вызовом конструктора перемещения типа `T`. Этот последний случай – единственный, когда мы фактически вызываем конструктор перемещения `T`!

Присваивание копированием следует похожей логике передачи экземпляра диспетчера из объекта справа, за исключением проверки трейта `propagate_on_container_copy_assignment`, или `pocca`.

Операция `swap` особенно интересна, потому что в ней заключительный случай (когда используется экземпляр диспетчера закрепляемого типа и два экземпляра – слева и справа – не равны) требует выполнить дополнительное распределение памяти:

```
void swap(uniqueish& rhs) noexcept {
  constexpr bool pocs =
    Traits::propagate_on_container_swap::value;
  using std::swap;
  if constexpr (pocs) {
    // Можно выполнить обмен диспетчерами, потому что
    // наш тип диспетчера "незакрепляемый".
    swap(this->m_allocator, rhs.m_allocator);
    swap(this->m_ptr, rhs.m_ptr);
  } else if (m_allocator == rhs.m_allocator) {
    // Наш диспетчер "закреплен" за этим контейнером;
    // но так как он эквивалентен диспетчеру в rhs,
    // мы можем принять память из rhs, и наоборот.
    swap(this->m_ptr, rhs.m_ptr);
  } else {
    // Ни одна из сторон не может принять память другой стороны,
    // и поэтому на той или другой стороне нужно вновь выделить память.
    auto temp = std::move(*this);
    *this = std::move(rhs); // может возбудить исключение
    rhs = std::move(temp);  // может возбудить исключение
  }
}
```

В обеих строках, отмеченных комментарием «может возбудить исключение», вызывается оператор присваивания перемещением, который в данном случае может вызвать метод `emplace` и обратиться к диспетчеру с просьбой

выделить память. Если соответствующий ему ресурс памяти окажется исчерпанным, `Traits::allocate(m_allocator, 1)` может возбудить исключение, и тогда мы окажемся перед лицом двух проблем. Во-первых, мы уже начали перемещение состояния и освободили старую память, и может так оказаться, что мы не сможем «вернуться» к прежнему состоянию. Во-вторых, что особенно важно, `swap` принадлежит к разряду настолько простых и фундаментальных функций, что стандартная библиотека не предусматривает возможности сбоя в ней. Например, алгоритм `std::swap` (глава 3 «Алгоритмы с парами итераторов») объявлен как `noexcept`, а это означает, что он всегда *должен* завершаться успехом; ему не позволено возбуждать исключения.

То есть если попытка выделить память потерпит неудачу в процессе выполнения функции `swap noexcept`, исключение `bad_alloc` начнет просачиваться вверх по стеку вызовов, пока не встретит объявление функции `swap noexcept`. В этой точке среда выполнения C++ прекратит раскрутку стека и вызовет функцию `std::terminate`, которая (если программист не изменил ее поведение с помощью `std::set_terminate`) вызовет аварийное завершение программы.

Стандарт C++17 пошел еще дальше в своем описании, что *должно* происходить в процессе обмена содержимого контейнеров стандартных типов. Во-первых, стандарт не утверждает, что ошибка выделения памяти в `swap` должна приводить к вызову `std::terminate`, он просто говорит, что ошибка в ходе выполнения `swap` приводит к *неопределенному поведению*. Во-вторых, стандарт не ограничивает неопределенное поведение ошибками распределения памяти! Согласно стандарту C++17, простой вызов `swap` для экземпляров любых стандартных контейнеров, диспетчеры памяти которых не равны, приводит в результате к неопределенному поведению, независимо от наличия или отсутствия ошибки распределения памяти!

Фактически libc++ использует эту возможность оптимизации, чтобы сгенерировать код для всех функций `swap` стандартных контейнеров, который выглядит примерно так:

```
void swap(uniqueish& rhs) noexcept {
  constexpr bool pocs =
    Traits::propagate_on_container_swap::value;
  using std::swap;
  if constexpr (pocs) {
    swap(this->m_allocator, rhs.m_allocator);
  }
  // Вообще не проверять - знаем ли мы как освободить
  // полученный указатель; просто предположить, что это возможно.
  swap(this->m_ptr, rhs.m_ptr);
}
```

Обратите внимание, что если использовать этот код (как это делает libc++) для обмена содержимого контейнеров с неравными диспетчерами памяти, вы столкнетесь с несоответствием между указателями и их диспетчерами, и ваша программа почти наверняка обрушится, или, что еще хуже, позднее вы можете

попытаться освободить один из указателей с помощью несоответствующего диспетчера. Помните об этой ловушке, используя «вспомогательные» типы C++17, такие как `std::pmr::vector`!

```
char buffer[100];
auto mr = std::pmr::monotonic_buffer_resource(buffer, 100);

std::pmr::vector<int> a {1,2,3};
std::pmr::vector<int> b({4,5,6}, &mr);

std::swap(a, b);
  // НЕОПЕРЕДЕЛЕННОЕ ПОВЕДЕНИЕ

a.reserve(a.capacity() + 1);
  // эта строка вызовет аварийное завершение,
  // так как попытается применить delete[] к указателю на стек
```

Если архитектура вашего кода позволяет обменивать содержимое контейнеров, хранящееся в разных ресурсах памяти, избегайте применения `std::swap` и пользуйтесь следующей безопасной идиомой:

```
auto temp = std::move(a); // OK
a = std::move(b); // OK
b = std::move(temp); // OK
```

Под словами «избегайте применения `std::swap`» я подразумеваю «избегайте любых перестановочных алгоритмов из STL», включая такие, как `std::reverse` и `std::sort`. Добиться правильной их работы можно, но это очень сложная задача, и я не советую браться за нее!

Если архитектура вашего кода позволяет обменивать содержимое контейнеров, хранящееся в разных ресурсах памяти, это верный признак, что ее стоит пересмотреть. Если вы сможете реорганизовать архитектуру так, что обмениваться будут только контейнеры, использующие общий ресурс памяти, или вам удастся избежать применения диспетчеров с состоянием и/или закрепляемых диспетчеров, тогда вы смело сможете позабыть об этой ловушке.

Передача вниз с scoped_allocator_adaptor

В предыдущем разделе я представил функцию `std::allocator_traits<A>::construct(a, ptr, args...)` и описал ее как предпочтительную альтернативу синтаксису «размещающей формы new» `::new((void*)ptr) T(args...)`. Теперь мы посмотрим, почему автору конкретного диспетчера памяти может потребоваться дать ей другую семантику.

Один из очевидных способов изменения семантики `construct` своего диспетчера памяти – сделать его тривиально простым типом с инициализацией по умолчанию вместо инициализации нулями. Вот как может выглядеть такой код:

```
template<class T>
struct my_allocator : std::allocator<T>
{
  my_allocator() = default;

  template<class U>
  my_allocator(const my_allocator<U>&) {}

  template<class... Args>
  void construct(T *p, Args&&... args) {
    if (sizeof...(Args) == 0) {
      ::new ((void*)p) T;
    } else {
      ::new ((void*)p) T(std::forward<Args>(args)...);
    }
  }
};
```

Теперь `std::vector<int, my_allocator<int>>` можно использовать как «векторный» тип, удовлетворяющий всем обычным ограничениям `std::vector<int>`, кроме случая неявного создания новых элементов посредством `v.resize(n)` или `v.emplace_back()`, когда новые элементы будут создаваться неинициализированными, подобно переменным на стеке.

То, что мы здесь создали, можно назвать «адаптером», который накладывается поверх `std::allocator<T>` и изменяет его поведение, как нам это нужно. Было бы еще лучше, если бы у нас была возможность аналогичным образом изменить, или «адаптировать», любой произвольный диспетчер памяти. И такая возможность имеется! Для этого нужно изменить наш `template<class T>` на `template<class A>` и наследовать `A` везде, где старый код наследует `std::allocator<T>`. Конечно, список шаблонных параметров адаптера больше не начинается с `T`, поэтому придется реализовать свою версию `rebind`. Однако этот путь ведет в густые дебри метапрограммирования, поэтому я не буду отвлекаться на его демонстрацию.

Но существует другой, более удобный подход к реализации метода `construct` для собственного типа диспетчера памяти. Взгляните на следующий пример, где создается вектор векторов целых чисел `int`:

```
std::vector<std::vector<int>> vv;
vv.emplace_back();
vv.emplace_back();
vv[0].push_back(1);
vv[1].push_back(2);
vv[1].push_back(3);
```

Допустим, мы решили «закрепить» этот контейнер за ресурсом памяти собственной разработки, таким как `WidgetAlloc`. Для этого нам нужно написать примерно такой код:

```
char buffer[10000];
std::pmr::monotonic_buffer_resource mr {buffer, sizeof buffer};

using InnerAlloc = WidgetAlloc<int>;
using InnerVector = std::vector<int, InnerAlloc>;
using OuterAlloc = WidgetAlloc<InnerVector>;

std::vector<InnerVector, OuterAlloc> vv(&mr);
vv.emplace_back(&mr);
vv.emplace_back(&mr);
vv[0].push_back(1);
vv[1].push_back(2);
vv[1].push_back(3);
```

Обратите внимание на повторение инициализатора `&mr` экземпляра диспетчера памяти на обоих уровнях. Необходимость повторения `&mr` усложняет использование нашего вектора `vv` в обобщенном контексте. Например, у нас не получится просто передать его в шаблон функции для заполнения данными, потому что для `emplace_back` нового вектора целых чисел `int` вызываемый код должен знать адрес `&mr`, известный только вызывающему. В этом случае можно создать обертку, воплощающую правило: «при создании нового элемента вектора векторов необходимо передать `&mr` в конце списка аргументов». И стандартная библиотека поможет нам в этом!

Начиная с версии C++11 стандартная библиотека предоставляет (в заголовке с именем `<scoped_allocator>`) шаблонный класс `scoped_allocator_adaptor<A>`. Так же как наш «адаптер» с инициализацией по умолчанию, `scoped_allocator_adaptor<A>` наследует `A`, заимствуя все черты поведения `A`; и затем переопределяет метод `construct`. В этом методе он пытается определить, конструируется ли объект `T` «с использованием диспетчера памяти», и если это так, передает себя конструктору `T` как дополнительный аргумент.

Выяснение, что тип `T` «использует диспетчера памяти», `scoped_allocator_adaptor<A>::construct`, производится с помощью типа `std::uses_allocator_v<T,A>`, который (если не специализирован вами, чего вы в большинстве случаев не должны делать) получит значение `true`, только если `A` неявно преобразуется в `T::allocator_type`. Если `T` не имеет `allocator_type`, тогда библиотека предположит, что `T` не заботится о выборе диспетчера памяти, кроме особых случаев `pair` и `tuple` (которые имеют специальные перегруженные версии конструкторов, предназначенные специально для передачи диспетчера вниз своим членам) и особого случая `promise` (который может выделять память для общего состояния даже притом, что не предусматривает никакого способа сослаться на объект диспетчера; мы говорим, что «стирание типов» в диспетчере памяти для `promise` выполняется еще тщательнее, чем в примерах, которые мы видели в главе 5 «Словарные типы»).

По исторически сложившимся причинам конструкторы типов с поддержкой диспетчеров памяти могут следовать одному из двух шаблонов программирования, и `scoped_allocator_adaptor` распознает оба. Более старые и простые ти-

пы (то есть все, кроме `tuple` и `promise`), как правило, имеют конструкторы вида: `T(args..., A)`, где диспетчер памяти `A` следует последним в списке. Для `tuple` и `promise` стандартная библиотека ввела новый шаблон: `T(std::allocator_arg, A, args...)`, где диспетчер `A` следует первым, но ему предшествует специальный тег `std::allocator_arg`, единственное назначение которого – показать, что следующий аргумент в списке аргументов представляет диспетчер памяти, подобно тому, как единственное назначение тега `std::nullopt` – показать, что `optional` не имеет значения (см. главу 5 «Словарные типы»). По аналогии с запретом создания типа `std::optional<std::nullopt_t>` стандарт также запрещает создавать `std::tuple<std::allocator_arg_t>`.

Используя `scoped_allocator_adaptor`, мы можем переписать наш громоздкий пример, сделав его менее громоздким:

```
char buffer[10000];
std::pmr::monotonic_buffer_resource mr {buffer, sizeof buffer};

using InnerAlloc = WidgetAlloc<int>;
using InnerVector = std::vector<int, InnerAlloc>;
using OuterAlloc =
std::scoped_allocator_adaptor<WidgetAlloc<InnerVector>>;

std::vector<InnerVector, OuterAlloc> vv(&mr);
vv.emplace_back();
vv.emplace_back();
vv[0].push_back(1);
vv[1].push_back(2);
vv[1].push_back(3);
```

Обратите внимание, что тип диспетчера памяти получился *более* громоздким, зато, что особенно важно, из вызовов `emplace_back` исчез аргумент `&mr`. Теперь мы можем использовать `vv` в контекстах, где применяется традиционный синтаксис добавления элементов, без передачи `&mr`. В нашем случае, поскольку мы используем свой диспетчер памяти, `WidgetAlloc`, не имеющий конструктора по умолчанию, симптомом забытого `&mr` являются ошибки времени компиляции. Но, как рассказывалось в предыдущих разделах в этой главе, `std::pmr::polymorphic_allocator<T>` благополучно позволит создавать его экземпляры с конструктором по умолчанию и потенциально разрушительными результатами; поэтому, если вы планируете использовать `polymorphic_allocator`, взгляните также на `scoped_allocator_adaptor`, просто чтобы ограничить количество мест, в которых вы могли бы забыть определить свою стратегию выделения памяти.

Передача разных диспетчеров памяти

В моем введении в `scoped_allocator_adaptor<A>` я не упомянул о еще одной сложности. Список параметров шаблона не ограничивается единственным аргументом с типом диспетчера памяти! Фактически можно создать тип,

перечислив несколько аргументов с типами диспетчеров, как в следующем примере:

```
using InnerAlloc = WidgetAlloc<int>;
using InnerVector = std::vector<int, InnerAlloc>;

using MiddleAlloc = std::scoped_allocator_adaptor<
  WidgetAlloc<InnerVector>,
  WidgetAlloc<int>
>;
using MiddleVector = std::vector<InnerVector, MiddleAlloc>;
using OuterAlloc = std::scoped_allocator_adaptor<
  WidgetAlloc<MiddleVector>,
  WidgetAlloc<InnerVector>,
  WidgetAlloc<int>
>;
using OuterVector = std::vector<MiddleVector, OuterAlloc>;
```

Определив все эти `typedef`, перейдем к созданию трех отдельных ресурсов памяти и сконструируем экземпляр `scoped_allocator_adaptor`, способный запомнить все три ресурса (потому что включает три отдельных экземпляра `WidgetAlloc`, по одному на «уровень»):

```
char bi[1000];
std::pmr::monotonic_buffer_resource mri {bi, sizeof bi};
char bm[1000];
std::pmr::monotonic_buffer_resource mrm {bm, sizeof bm};
char bo[1000];
std::pmr::monotonic_buffer_resource mro {bo, sizeof bo};

OuterAlloc saa(&mro, &mrm, &mri);
```

Наконец, можно сконструировать экземпляр `OuterVector`, передав аргумент `scoped_allocator_adaptor`, и все! Переопределенный метод `construct`, глубоко скрытый в нашем тщательно подготовленном типе диспетчера, позаботится о передаче аргумента `&bm` или `&bi` в любой конструктор, нуждающийся в них:

```
OuterVector vvv(saa);

vvv.emplace_back();
  // В этом случае память будет зарезервирована в буфере "bo".

vvv[0].emplace_back();
  // В этом случае память будет зарезервирована в буфере "bm".

vvv[0][0].emplace_back(42);
  // В этом случае память будет зарезервирована в буфере "bi".
```

Как видите, глубоко вложенный `scoped_allocator_adaptor` – не для слабых духом и имеет практическую ценность, только если определить множество «вспомогательных» определений `typedef`, как в этом примере.

И еще одно замечание о `std::scoped_allocator_adaptor<A...>`: Если вложенность контейнеров глубже, чем количество типов диспетчеров в списке параметров шаблона, тогда `scoped_allocator_adaptor` будет действовать так, как если бы последний тип диспетчера в списке параметров повторялся до бесконечности. Например:

```
using InnerAlloc = WidgetAlloc<int>;
using InnerVector = std::vector<int, InnerAlloc>;

using MiddleAlloc = std::scoped_allocator_adaptor<
  WidgetAlloc<InnerVector>
>;
using MiddleVector = std::vector<InnerVector, MiddleAlloc>;

using TooShortAlloc = std::scoped_allocator_adaptor<
  WidgetAlloc<MiddleVector>,
  WidgetAlloc<InnerVector>
>;
using OuterVector = std::vector<MiddleVector, TooShortAlloc>;

TooShortAlloc tsa(&mro, WidgetAlloc<InnerVector>(&mri));
OuterVector tsv(tsa);

tsv.emplace_back();
  // В этом случае память будет зарезервирована в буфере "bo".

tsv[0].emplace_back();
  // В этом случае память будет зарезервирована в буфере "bi".

tsv[0][0].emplace_back(42);
  // В этом случае память СНОВА будет зарезервирована в буфере "bi"!
```

Фактически именно на такое поведение мы положились в нашем первом примере использования `scoped_allocator_adaptor` для создания вектора векторов `vv`, даже притом, что тогда я не упомянул об этом явно. Теперь, когда вы познакомились с ним, можете вернуться назад и исследовать пример еще раз, чтобы заметить «повторение до бесконечности» и как можно изменить тот код, чтобы для внутреннего вектора целых чисел использовать другой ресурс памяти, отличный от ресурса, используемого для внешнего вектора `InnerVector`.

Итоги

Диспетчеры памяти являются самой загадочной темой в C++, в основном по исторически сложившимся причинам. Существует несколько разных интерфейсов с малопонятными случаями использования, накладывающимися друг на друга. Все они тесно связаны с метапрограммированием, и все еще отсутствует поддержка многих особенностей, даже относительно старых, относящихся к стандарту C++11, таких как причудливые указатели.

Стандарт C++17 предложил библиотечный тип `std::pmr::memory_resource`, чтобы более четко провести грань между *ресурсами памяти* (или кучами) и *диспетчерами памяти* (или дескрипторами куч). Ресурсы памяти поддерживают методы `allocate` и `deallocate`; диспетчеры памяти тоже предлагают эти методы, а также методы `construct` и `destroy`.

Решив реализовать свой тип диспетчера памяти `A`, вы должны объявить его шаблоном; первым параметром этого шаблона должен быть тип `T` объектов, для которых предполагается выделять память. Кроме того, ваш тип диспетчера `A` должен иметь шаблонный конструктор для поддержки «перепривязки» из `A<U>` в `A<T>`. Так же как любой указатель, тип диспетчера памяти должен поддерживать операторы `==` и `!=`.

Метод кучи `deallocate` может требовать передачи дополнительных метаданных, присоединенных к входному указателю. В C++ эта возможность поддерживается посредством *причудливых указателей*. Тип `std::pmr::memory_resource` в C++17 не поддерживает причудливых указателей, но вы легко сможете реализовать свою поддержку.

Типы причудливых указателей должны удовлетворять всем требованиям, которые предъявляются к итераторам с произвольным доступом, поддерживать пустые значения и давать возможность преобразовывать их в обычные указатели. Если вы решите использовать свой тип причудливых указателей с контейнерами на основе узлов, такими как `std::list`, вы должны реализовать статическую функцию-член `pointer_to`.

C++17 различает типы «закрепляемых» и «незакрепляемых» диспетчеров памяти. Типы диспетчеров без состояния, такие как `std::allocator<T>`, являются незакрепляемыми; типы диспетчеров с состоянием, такие как `std::pmr::polymorphic_allocator<T>`, по умолчанию являются закрепляемыми. Чтобы получить свой тип диспетчера памяти с нестандартным свойством закрепляемости или незакрепляемости, нужно добавить в него все три члена `typedef`, известные как «POCCA», «POCMA» и «POCS». Типы закрепляемых диспетчеров, такие как `std::pmr::polymorphic_allocator<T>`, в основном используются в традиционном объектно-ориентированном программировании, когда объект контейнера закрепляется за конкретным адресом памяти. Типы диспетчеров без состояния в основном применяются в программировании с использованием семантики значений (со множеством операций `move` и `swap`), а также везде, где программа использует одну кучу и один закрепляемый диспетчер памяти, фактически не имеющий состояния.

Тип `scoped_allocator_adaptor<A...>` может помочь упростить использование глубоко вложенных контейнеров, использующих нестандартные ресурсы памяти или диспетчеры памяти. Почти любой глубоко вложенный контейнер, использующий нестандартный тип диспетчера памяти, требует определения множества вспомогательных типов в ущерб удобочитаемости кода.

Обмен местами двух контейнеров с разными закрепляемыми диспетчерами памяти в теории порождает неопределенное поведение, но на практике повреждает данные в памяти и вызывает аварийное завершение. Не делайте этого!

Глава 9

Потоки ввода/вывода

До сих пор мы видели классический полиморфизм в стандартной библиотеке лишь в нескольких местах. В главе 8 «Диспетчеры памяти» мы познакомились с классически полиморфным типом `std::pmr::memory_resource`, и в главе 5 «Словарные типы» видели, как «за кулисами» классический полиморфизм используется типами `std::any` и `std::function`. Однако в большей своей части стандартная библиотека благополучно обходится без классического полиморфизма.

Но в двух местах классический полиморфизм используется очень широко. Одно из них – иерархия стандартных исключений (для большего удобства все исключения, возбуждаемые стандартной библиотекой, наследуют класс `std::exception`), но в этой книге мы не будем рассматривать иерархию исключений. Другое – стандартный заголовок `<iostream>`, который рассматривается в этой главе. Однако мы должны прояснить еще кое-что, прежде чем углубиться в исследование этого заголовка!

В этой главе рассматриваются следующие темы:

- разделение вывода на буферизацию и форматирование и разделение ввода на буферизацию, лексемизацию и парсинг;
- POSIX API для неформатированного ввода/вывода в файлы;
- «C» API в `<stdio.h>`, добавляющий буферизацию и форматирование;
- достоинства и недостатки классического API `<iostream>`;
- опасности форматирования *в зависимости от региональных настроек* и новые возможности C++17, которые помогут избежать их;
- множество способов преобразования числовых данных в строки и обратно.

Проблемы ввода/вывода в C++

Типичной мерой удобства языков программирования является степень сложности программ типа «Hello world». Многие популярные языки программирования имеют очень низкий показатель «Hello world»: в большинстве языков сценариев, таких как Python и Perl, программа «Hello world» состоит буквально из одной строки: `print "hello world"`.

C++ и его предшественник C являются языками системного программирования, в первую очередь ориентированными на управление машиной, высокую скорость выполнения и (в случае с C++) возможность использовать систему типов для реализации обобщенных алгоритмов. В это разнообразие целей не вписываются такие маленькие программы, как «Hello world».

Каноническая программа «Hello world» на языке C имеет следующий вид:

```
#include <stdio.h>

int main()
{
  puts("hello world");
}
```

В C++:

```
#include <iostream>

int main()
{
  std::cout << "hello world" << std::endl;
}
```

Канонический исходный код на C++ не особенно длиннее канонического исходного кода на C, но имеет намного больше «параметров» настройки, с которыми должен познакомится любой начинающий программист, даже если не собирается изменять их. Например, там, где в C мы вызываем функцию `puts` (фактически «глагол»), в C++ мы применяем *оператор* к объекту `std::cout` (фактически здесь мы имеем «глагол» и «косвенное дополнение»). Чтобы написать пример на C++, мы также должны предварительно познакомится со специальным именем символа конца строки – `std::endl` – тонкостью, которую функция `puts` в C скрывает от нас.

Иногда такая сложность отпугивает новичков от C++, особенно если они учат C++ в школе и не имеют возможности выбора. Однако это всего лишь досадное недоразумение! Предыдущий исходный код на «C» (с использованием `puts`) является также допустимым кодом на C++, и в использовании инструментов из `<stdio.h>` нет ничего неправильного. Фактически в этой главе мы сначала познакомимся поближе с возможностями в `<stdio.h>` и только потом перейдем к `<iostream>`. Однако мы увидим, что C++14 и C++17 ввели несколько малоизвестных функций – в таких заголовках, как `<string>` и `<utility>`, – которые упрощают решение многих задач ввода/вывода.

Примечание об именах заголовков: для обозначения заголовка со средствами ввода/вывода «в стиле C» я использую имя `<stdio.h>`. Начина с C++03 появился похожий стандартный заголовок `<cstdio>`. Единственное различие между `<stdio.h>` и `<cstdio>` состоит в том, что все инструменты, объявленные в `<stdio.h>`, гарантированно доступны в глобальном пространстве имен (например, `::printf`) и могут быть доступны или недоступны в пространстве

имен `std` (например, `std::printf`); тогда как инструменты, объявленные в `<cstdio>`, гарантированно доступны в `std` (например, `std::printf`), но могут быть доступны или недоступны в глобальном пространстве имен (например, `::printf`). На практике между ними нет никакой разницы, потому что все ведущие создатели реализаций помещают эти инструменты в оба пространства имен, и они остаются доступными при подключении любого их этих заголовков. Лично я советую выбрать для себя какой-то один стиль и придерживаться его. Если в вашем проекте используется большое количество заголовков POSIX, таких как `<unistd.h>`, имена которых всегда оканчиваются на `.h`, эстетически может быть предпочтительнее использовать стандартные заголовки с именами на `.h` «в стиле C».

Буферизация и форматирование

Разобраться в хитросплетениях ввода/вывода «в стиле C» и в «стиле iostream» будет проще, если вспомнить, что при вводе (так же как и при выводе) выполняются две фундаментально разные операции. Чтобы хоть как-то обозначить их, дадим им имена: *буферизация* и *форматирование*.

- *Форматирование* заключается в получении от программы строго типизированных данных – целых чисел, строк, чисел с плавающей точкой, экземпляров пользовательских классов – и их преобразовании (сериализации) в «текст». Например, число 42 выводится как `"42"` (или `"+42"`, или `"0x002A"`), то есть *форматируется*. Обычно библиотека форматирования реализует свой «мини-язык» для описания деталей форматирования каждого значения.
- *Буферизация* заключается в получении от программы пакета двоичных байтов и отправке их в некоторое устройство вывода (на выход) или в извлечении данных из некоторого устройства ввода и передаче программе в виде пакета байтов (на вход). Часть библиотеки, решающая проблемы буферизации, может, например, «собирать данные в пакеты по 4096 байт и затем выталкивать их» или обслуживать объекты ввода/вывода: файлы в файловой системе, сетевые сокеты или массивы байтов в памяти.

Я сознательно употребил слова «текст», описывая выход стадии *форматирования*, и «пакет байтов», описывая вход стадии *буферизации*. В традиционных операционных системах «текст» и «байты» – суть одно и то же. Но если вы пользуетесь одной из тех странных операционных систем, где окончания строк кодируются двумя байтами или где для текстовых файлов используются кодировки, отличные от UTF-8, тогда в одной или в обеих стадиях, или ниже по течению (как, например, в некоторых системах, где системный вызов записывает данные в файл), должна иметь место дополнительная обработка. Мы не будем больше говорить о подобных вещах, потому что, как я надеюсь, вы не используете такие операционные системы (или региональные настройки) в промышленном окружении. В промышленном окружении всегда следует ис-

пользовать кодировку символов UTF-8, часовой пояс UTC и выбирать региональные настройки `"C.UTF-8"`. Поэтому для целей дальнейшего обсуждения будем полагать, что «форматирование» и «буферизация» – единственные этапы конвейера, которые представляют для нас интерес.

Выполняя ввод, мы сначала производим «буферизацию», чтобы прочитать байты из устройства ввода; а затем «форматируем» их, превращая в строго типизированные данные. Этап «форматирования» для ввода может делиться на *лексемизацию* (выделение отдельных элементов данных в потоке) и *парсинг* (преобразование байтов в фактические данные). Подробнее о лексемизации мы поговорим в главе 10 «Регулярные выражения».

POSIX API

Самое главное, о чем нужно помнить, говоря о вводе/выводе в файлы, – все, что связано с вводом/выводом в C и C++, основано на стандарте POSIX. POSIX – это очень низкоуровневая спецификация, почти на уровне системных вызовов Linux, почти не пересекающаяся со стандартными API ввода/вывода в C и C++; и, если не разобраться в сути уровня POSIX, будет очень сложно понять идеи, которые описываются далее.

Имейте в виду, что практически *ни один* из API, описываемых ниже, не определяется стандартом C++! Это, скорее, допустимый C++, который соответствует стандарту POSIX, *не являющемуся стандартом C++*. С практической точки зрения это означает, что эти API будут работать в любой операционной системе, кроме Windows, и, может быть, будут работать в современных версиях Windows через слой **Windows Subsystem for Linux** (WSL). Но, как бы то ни было, все стандартные API (`<stdio.h>` и `<iostream>`) основаны на этой модели.

Большую часть из API, описываемых ниже, определяют нестандартные заголовки `<unistd.h>` и `<fcntl.h>`.

В POSIX под термином *файл* понимается фактический файл на диске (или, по крайней мере, в *файловой системе* некоторого рода; простите меня, если я иногда буду использовать слово «диск», подразумевая файловую систему). Один и тот же файл может конкурентно использоваться сразу несколькими программами, посредством ресурсов операционной системы, известных как *файловые дескрипторы*. В программах на C или C++ вам никогда не придется работать непосредственно с объектами файловых дескрипторов; вместо них вы будете пользоваться ссылками (или указателями) на файловые дескрипторы. Эти ссылки (или указатели) представляют себя не как указатели в классическом понимании, а как маленькие целые числа – буквально значения типа `int`. (Комитет по разработке стандарта POSIX не так одержим безопасностью типов, как средний программист на C++!)

Для создания новых файловых дескрипторов и получения целочисленных ссылок на них используется функция `open`; например, `int fd = open("myfile.txt", O_RDONLY)`. Второй аргумент – это битовая маска, которая может содержать любые из следующих флагов, объединяемых по ИЛИ:

- **обязательные:** один и только один «режим доступа». К допустимым «режимам доступа» относятся `O_RDONLY` (только для чтения), `O_WRONLY` (только для записи) и `O_RDWR` (для чтения и для записи);
- **необязательные:** некоторые «флаги времени открытия файла», описывающие действия, которые должна выполнить система в момент открытия файла. Например, `O_CREAT` означает: «если указанный файл отсутствует, создать его» (в противоположность возврату признака ошибки). Также можно добавить флаг `O_EXCL`, который означает: «...а если указанный файл существует, вернуть признак ошибки». Еще один важный флаг времени открытия – `O_TRUNC`, который означает: «усечь – очистить, опустошить, стереть – файл после открытия»;
- **необязательные:** некоторые «режимы работы», описывающие способ выполнения операций ввода/вывода с этим файловым дескриптором. Наиболее важным из них является флаг `O_APPEND`.

Флаг `O_APPEND` обозначает «режим добавления в конец». Когда для работы с файлом используется «режим добавления в конец», читать данные можно из любого места в файле (как будет показано далее), но перед каждой операцией записи неявно выполняется операция перехода в конец файла (то есть после записи текущая позиция ссылается на конец файла, даже если перед этим вы читали данные из другого места). Режим добавления в конец удобно использовать для журналирования, особенно когда запись может производиться из нескольких потоков выполнения. Некоторые стандартные служебные программы, такие как `logrotate`, успешнее справляются со своими обязанностями, когда программы, осуществляющие журналирование, правильно открывают свои файлы журналов в «режим добавления в конец». Проще говоря, режим добавления в конец используется настолько широко, что он снова и снова находит применение в каждом API высокого уровня.

Теперь разберемся с понятиями «текущей позиции» и «позиционирования». С каждым файловым дескриптором POSIX связаны некоторые данные – по сути, это «переменные-члены». Один из таких членов данных описывает режим работы дескриптора; другой определяет *текущую позицию в файле*, в дальнейшем я буду называть его «курсором». Подобно курсору в текстовом редакторе, этот курсор указывает на место внутри файла, с которого начнется следующая операция чтения или записи. Операции чтения и записи с дескриптором передвигают его курсор вперед. И, как говорилось в предыдущем абзаце, выполнение операции записи с дескриптором, действующим в «режиме добавления в конец», автоматически переносит курсор в самый конец файла. Обратите внимание, что в дескрипторе имеется только один курсор! Открыв файловый дескриптор с флагом `O_RDWR`, вы получите не два курсора – по одному для чтения и для записи, а только один, который перемещается *и* операциями записи, *и* операциями чтения.

`read(fd, buffer, count)`: читает байты из файла и сохраняет их в буфере `buffer`, продолжая чтение, пока не будет получено `count` байтов или не встре-

тится временная или постоянная ошибка (например, если необходима пауза, чтобы получить больше данных из сетевого соединения, или файловая система с файлом была отмонтирована в середине операции чтения). Возвращает количество прочитанных байтов и, как вы помните, передвигает курсор вперед.

`write(fd, buffer, count)`: записывает байты из буфера `buffer` в файл, пока не запишет `count` байтов или не встретится временная или постоянная ошибка. Возвращает количество записанных байтов; и, как вы помните, передвигает курсор вперед. (А кроме того, если файловый дескриптор открыт в режиме добавления в конец, перед записью любых данных передвигает курсор в конец файла.)

`lseek(fd, offset, SEEK_SET)`: позиционирует (то есть передвигает) курсор с заданным смещением от начала файла и возвращает это смещение (или `-1`, если операция потерпела неудачу, например встретив конец файла).

`lseek(fd, offset, SEEK_CUR)`: позиционирует курсор с заданным смещением относительно *текущей* позиции курсора. Относительные перемещения редко используются на практике, но вызов `lseek(fd, 0, SEEK_CUR)` занимает особое место – именно так можно узнать текущую позицию курсора!

`lseek(fd, offset, SEEK_END)`: позиционирует курсор с заданным смещением от конца файла. И снова эта версия наиболее полезна, когда смещение `offset` равно нулю.

Кстати, не существует способа, с помощью которого можно было бы создать копию POSIX-дескриптора файла, чтобы получить второй курсор в том же файле. Чтобы получить два курсора, вам придется открыть файл дважды. По иронии существует POSIX-функция с именем `dup`, которая принимает целочисленную ссылку на файловый дескриптор и возвращает другую целочисленную ссылку на тот же дескриптор; это своего рода примитивный механизм подсчета ссылок.

По окончании работы с файловым дескриптором вызывается функция `close(fd)`, освобождающая ссылку; если это была последняя ссылка на дескриптор (то есть если никакой другой код в программе не вызвал `dup`), тогда сам файловый дескриптор тоже освобождается и возвращается операционной системе – то есть сам файл в файловой системе «закрывается».

А теперь объединим все вместе и напишем простую программу, использующую POSIX API, чтобы открыть, прочитать, записать, изменить позицию курсора и закрыть файл:

```
#include <cassert>
#include <string>
#include <unistd.h>
#include <fcntl.h>

int main()
{
  int fdw = open("myfile.txt", O_WRONLY | O_CREAT | O_TRUNC);
  int fdr = open("myfile.txt", O_RDONLY);
```

```
  if (fdw == -1 || fdr == -1)
    return EXIT_FAILURE;

  write(fdw, "hello world", 11);
  lseek(fdw, 6, SEEK_SET);
  write(fdw, "neighbor", 8);

  std::string buffer(14, '\0');
  int b = read(fdr, buffer.data(), 14);
  assert(b == 14);
  assert(buffer == "hello neighbor");
  close(fdr);
  close(fdw);
}
```

Обратите внимание, что POSIX API не предусматривает ничего, что было бы связано с *форматированием*. Он просто дает возможность прочитать байты с диска или записать их на диск. Аналогично POSIX API не предусматривает ничего, что было бы связано с «буферизацией вывода»: когда вызывается `write`, она просто записывает ваши данные. То есть данные могут сохраняться в буфере на уровне операционной системы, контроллера диска или оборудования, но с точки зрения программы данные *уже записаны*. Любые задержки в фактической записи вам неподвластны и не являются вашей ошибкой. Это, в свою очередь, означает, что при необходимости эффективно записать большой объем данных с использованием POSIX API ваша программа должна сама организовать буферизацию вывода данных и затем посылать буфер целиком в единственный вызов `write`. Единственная операция записи 4096-байтного блока выполнится быстрее, чем 4096 записей по одному байту!

Или, чтобы не писать свой код управления буферизацией, можно подняться уровнем выше и использовать C API.

Стандартный C API

Описание этого API будет таким же кратким и неполным, как предыдущее обсуждение POSIX. За более полным описанием инструментов в `<stdio.h>` вам придется обратиться к другому источнику, например к *cppreference.com* или страницам справочного руководства `man`.

В C API *файловые дескрипторы* POSIX получили новое имя: аналог файлового дескриптора называется `FILE`, а аналог целочисленной ссылки на дескриптор – естественно, `FILE*`. Однако так же, как в POSIX API, вы не сможете сконструировать экземпляр `FILE` вручную.

Чтобы создать новый объект `FILE` и получить указатель на него, нужно вызвать функцию `fopen`, например `FILE *fp = fopen("myfile.txt", "r")`. Второй аргумент – это строка (то есть указатель на массив символов, завершающийся нулем, но на практике чаще используются строковые литералы, как здесь), которая может быть одной из следующих:

- `"r"`: эквивалент POSIX-флага `O_RDONLY`. Открывает файл для чтения. Если файл отсутствует, возвращается признак ошибки (то есть `nullptr`);
- `"w"`: эквивалент комбинации POSIX-флагов `POSIX O_WRONLY | O_CREAT | O_TRUNC`. Открывает файл для записи. Создает файл, если он отсутствует. Перед выполнением любых других операций файл опустошается;
- `"r+"`: эквивалент комбинации POSIX-флагов `O_RDWR | O_CREAT`. Открывает файл для чтения и для записи. Создает файл, если он отсутствует;
- `"w+"`: эквивалент комбинации POSIX-флагов `O_RDWR | O_CREAT | O_TRUNC`. Открывает файл для чтения и для записи. Создает файл, если он отсутствует. Перед выполнением любых других операций файл опустошается;
- `"a"`: эквивалент комбинации POSIX-флагов `O_WRONLY | O_CREAT | O_APPEND`. Открывает файл для записи. Создает файл, если он отсутствует. Устанавливает режим добавления в конец;
- `"a+"`: эквивалент комбинации POSIX-флагов `O_RDWR | O_CREAT | O_APPEND`. Открывает файл для чтения и для записи. Создает файл, если он отсутствует. Устанавливает режим добавления в конец.

Обратите внимание, что в предыдущих строках наблюдается один и тот же шаблон – символ `'+'` всегда отображается в флаг `O_RDWR`, символ `'w'` – в флаг `O_TRUNC`, а символ `'a'` – в `O_APPEND`; однако нет такого же регулярного шаблона, описывающего отображение строк режимов `fopen` в POSIX-флаги открытия файла.

Некоторые платформы поддерживают добавление дополнительных символов в конец строки режима; например, во многих POSIX-платформах можно добавить символ `'x'`, соответствующий флагу `O_EXCL`; на платформах GNU поддерживается символ `'e'`, соответствующий флагу `O_CLOEXEC`; а в Windows аналогичное поведение дает дополнительный символ `'N'`.

Единственный символ, который можно добавить в строку режима на любой платформе (он гарантируется стандартом C++ и доступен повсюду) – `'b'`, от англ. «binary» (двоичный). Это имеет значение только для Windows, где, если не указать этот символ, библиотека автоматически будет преобразовывать каждый выводимый байт `'\n'` в последовательность завершения строки `"\r\n"`, принятую в Windows. Если вам действительно нужно такое преобразование при работе в Windows, используйте удобное соглашение – добавляйте символ `'t'` в строку режима. Все реализации библиотеки распознают и игнорируют этот символ; он просто служит индикатором, что программист действительно хотел открыть файл в «текстовом» режиме и не упустил символ `'b'` по оплошности.

Закончив работу с файлом, вы должны вызвать функцию `fclose(fp)`, соответствующую функции `close(fd)`, которая закроет ссылку на файловый дескриптор.

Для автоматического выполнения операций с указателями `FILE` в стиле C можно использовать умные указатели, о которых рассказывалось в главе 5

«Словарные типы». Вот, например, как можно реализовать «уникальный указатель FILE»:

```
struct fcloser {
  void operator()(FILE *fp) const {
    fclose(fp);
  }

  static auto open(const char *name, const char *mode) {
    return std::unique_ptr<FILE, fcloser>(fopen(name, mode));
  }
};

void test() {
  auto f = fcloser::open("test.txt", "w");
  fprintf(f.get(), "hello world\n");
    // f будет закрыт автоматически
}
```

Также напомню, что `unique_ptr` всегда можно переместить в `shared_ptr`, если возникнет потребность организовать подсчет ссылок по принципу: «последний уходящий выключает свет»:

```
auto f = fcloser::open("test.txt", "w");
std::shared_ptr<FILE> g1 = std::move(f);
  // теперь f - пустой указатель, а счетчик в g1 получает значение 1
if (true) {
  std::shared_ptr<FILE> g2 = g1;
    // счетчик в g1 теперь равен 2
  fprintf(g2.get(), "hello ");
    // счетчик в g1 уменьшается до 1
}
fprintf(g1.get(), "world\n");
  // счетчик ссылок в g1 уменьшается до 0; файл закрывается
```

Буферизация в стандартном C API

Стандартный C API предлагает семейство функций, напоминающих POSIX-функции, но с именами, начинающимися с `f`.

Функция `fread(buffer, 1, count, fp)` читает из файла и сохраняет в указанном буфере `buffer` до `count` байтов или пока не столкнется с некоторой постоянной ошибкой (например, если кто-то отмонтировал файловую систему в процессе чтения). Она возвращает число прочитанных байтов и передвигает курсор.

Литерал `1` в этом вызове – не ошибка! Строго говоря, функция имеет сигнатуру `fread(buffer, k, count, fp)`. Она читает до `k * count` байтов (или пока не столкнется с некоторой постоянной ошибкой), и возвращает число прочитанных байтов, деленное на `k`. Однако в подавляющем большинстве случаев в параметре `k` передается литерал `1`; использование любого другого значения

считается ошибкой по двум основным причинам. Во-первых, поскольку возвращаемое значение всегда делится на k, может произойти потеря информации, если k будет иметь любое другое значение, кроме 1. Например, если в k передать 8, тогда возвращаемое значение 3 будет означать: было прочитано и сохранено в буфер «где-то между 24 и 31» байт, и элемент buffer[3] может содержать неполное значение – то есть фактически мусор – и у вас не будет никакой возможности определить этот факт. Во-вторых, поскольку внутренне библиотека умножает k * count, передача в параметре k любого значения, отличного от 1, увеличивает риск переполнения буфера и неправильного определения его размера. Ни одна популярная реализация не проверяет факт переполнения; это делается исключительно ради высокой производительности. Нет смысла тратить драгоценное процессорное время на дорогостоящую операцию деления, если каждый программист уже знает и понимает, что в k не следует передавать никаких других значений, кроме 1!

Функция fwrite(buffer, 1, count, fp) записывает из буфера buffer в файл до count байтов или пока не столкнется с некоторой постоянной ошибкой. Она возвращает число записанных байтов и передвигает курсор. (И если файл открыт в режиме записи в конец, *перед* записью передвигает курсор в конец файла.)

Функция fseek(fp, offset, SEEK_SET) устанавливает текущую позицию (то есть передвигает курсор) с указанным смещением от начала файла; fseek(fp, offset, SEEK_CUR) устанавливает текущую позицию с заданным смещением относительно *текущего* курсора; а fseek(fp, offset, SEEK_END) устанавливает текущую позицию с заданным смещением относительно конца файла. В отличие от POSIX-функции lseek, стандартная C-версия fseek не возвращает текущего значения курсора; она просто возвращает 0 в случае успеха и –1 в случае ошибки.

Функция ftell(fp) возвращает текущее значение курсора; то есть она действует подобно вызову POSIX-функции lseek(fd, 0, SEEK_CUR).

Кстати, коль скоро зашла речь о POSIX-функциях: если вы пишете программу для платформы, поддерживающей стандарт а POSIX, и вам нужно сделать что-то нестандартное с файловым дескриптором POSIX, на который ссылается стандартный указатель FILE *, вы можете получить дескриптор вызовом fileno(fp). Например, вот как можно выразить функцию ftell:

```
long ftell(FILE *fp)
{
  int fd = fileno(fp);
  return lseek(fd, 0, SEEK_CUR);
}
```

В целом можно работать с функциями fread и fwrite, но это не самый распространенный способ использования C API. Многие программисты предпочитают производить ввод и вывод не блоками данных, а посимвольно или побайтно. Оригинальная «философия Unix» ориентирована на применение

простых утилит командной строки, которые читают и преобразуют «поток» байтов; такие программы известны как «фильтры», и они проявляют себя во всем блеске, когда объединяются в конвейеры командной оболочки Unix. Например, вот маленькая программка, открывающая файл и подсчитывающая количество байтов, разделенных пробелами «слов» и строк, используя `<stdio.h>`:

```
struct LWC {
  int lines, words, chars;
};

LWC word_count(FILE *fp)
{
  LWC r {};
  bool in_space = true;
  while (true) {
    int ch = getc(fp);
    if (ch == EOF) break;
    r.lines += (ch == '\n');
    r.words += (in_space && !isspace(ch));
    r.chars += 1;
    in_space = isspace(ch);
  }
  return r;
}

int main(int argc, const char **argv)
{
  FILE *fp = (argc < 2) ? stdin : fopen(argv[1], "r");
  auto [lines, words, chars] = word_count(fp);
  printf("%8d %7d %7d\n", lines, words, chars);
}
```

(Узнали? Это утилита командной строки `wc`.)

Эта программа демонстрирует две новые идеи (кроме того что стандарт гарантирует неявное закрытие всех объектов `FILE` по завершении программы, благодаря чему можно безопасно завершать программу, не вызывая `fclose` и экономя несколько строк кода). Первое – программа использует *стандартные потоки ввода/вывода*. В C и C++ поддерживается три стандартных потока ввода/вывода: `stdin`, `stdout` и `stderr`. В своей программе подсчета слов мы следуем правилу: если пользователь не указал явно имя файла для чтения, ввод данных производится из `stdin`, стандартного потока ввода, который обычно является синонимом консоль (или терминала, или клавиатуры). В операционной системе и командной оболочке имеются разнообразные механизмы, которые можно использовать для перенаправления других источников входных данных в поток ввода; мы не будем рассматривать эти механизмы (такие как `wc <myfile.txt`), потому что их обсуждение выходит за рамки книги. Главное, что нужно помнить о трех стандартных потоках ввода/вывода, – они автоматически до-

ступны по своим именам, для них не требуется вызывать fopen, и попытка закрыть любой из них вызовом fclose всегда приводит к ошибке.

Вторая новая идея, представленная в программе подсчета слов, – функция getc. Функция getc(fp) читает один байт из указанного FILE * и возвращает его. Если возникла ошибка или (что более вероятно) достигнут конец файла, она возвращает специальное значение с именем EOF. Обычно EOF имеет числовое значение -1, но при этом гарантируется, что оно всегда отличается от любого значения, допустимого для действительного байта. По этой причине возвращаемое значение функции getc(fp) имеет тип int, а не char, то есть достаточно большой, чтобы хранить все возможные значения char, а также отличающееся от них значение EOF (если в вашей системе тип char поддерживает знак – как на многих платформах, – тогда getc преобразует char в unsigned char перед возвратом вызывающей программе; благодаря этому байт 0xFF будет представлен числом 255, которое явно отличается от значения –1, представляющего EOF).

Теперь отметим существенные отличия между fread/fwrite и read/write. Напомню, что POSIX API не предусматривает никакой дополнительной буферизации ни для ввода, ни для вывода; вызывая read, вы напрямую обращаетесь к операционной системе с просьбой вернуть следующий блок байтов. Если бы getc(fp) была реализована как fread(&ch, 1, 1, fp), а fread(buf, 1, count, fp) – как read(fileno(fp), buf, count), наша программа подсчета слов оказалась бы чудовищно неэффективной – для чтения содержимого файла, содержащего миллион байт, потребовалось бы выполнить миллион системных вызовов! Поэтому, обертывая файловый дескриптор объектом FILE, библиотека C добавляет еще одну особенность: *буферизацию*.

Потоки FILE могут быть «небуферизованными» (в том смысле, что каждый вызов fread действительно соответствует вызову read, а каждый вызов fwrite – вызову write); «полностью буферизованными», или «блочно-буферизованными» (когда записываемые данные накапливаются в скрытом буфере и посылаются в дескриптор файла после заполнения этого буфера; аналогично чтение производится из скрытого буфера, который заполняется новыми данными из дескриптора после опустошения); или «построчно буферизованными» (когда также имеется скрытый буфер, но он выталкивается в дескриптор не только по заполнении, но также при записи символа '\n'). Когда программа запускается и открывает стандартные потоки ввода/вывода, stdin и stdout переводятся в режим построчной буферизации, а для stderr буферизация полностью отключается. Любые файлы, которые вы открываете сами вызовом fopen, обычно получают режим полной буферизации, при этом операционная система может изменить режим буферизации; например, если открываемый вами «файл» фактически является устройством терминала, система может открыть его в режиме построчной буферизации.

В редких случаях бывает желательно явно указать режим буферизации для потока FILE. Для этой цели существует стандартная функция setvbuf. С ее помощью также можно определить свой буфер, как показано ниже:

```
FILE *fp = fopen("myfile.txt", "w");
  int fd = fileno(fp);
  char buffer[150];
  setvbuf(fp, buffer, _IOFBF, 150);
    // setvbuf возвращает 0 в случае успеха и EOF в случае ошибки.

  std::string AAAA(160, 'A');
  int bytes_written = fwrite(AAAA.data(), 1, 160, fp);
    // Этот вызов запишет в буфер 150 байт, вытолкнет его
    // и запишет в буфер еще 10 байт.

  assert(bytes_written == 160);
  assert(lseek(fd, 0, SEEK_CUR) == 150);
  assert(ftell(fp) == 160);
```

Обратите внимание на несоответствие между `ftell(fp)` и `lseek(fd, 0, SEEK_CUR)` в двух последних строках примера. Десять байт продолжают оставаться в буфере `buffer`, поэтому `FILE` сообщит, что в данный момент курсор имеет смещение 160, но в действительности курсор в дескрипторе POSIX все еще имеет смещение 150, и продолжит оставаться с этим смещением, пока буфер `buffer` не заполнится и не будет вытолкнут во второй раз – в этот момент курсор в дескрипторе POSIX переместится в позицию со смещением 300. Это несколько неудобно, но на самом деле это именно то, что нам нужно! Нам нужна эффективность, которая достигается за счет записи данных в дескриптор большими блоками. (Обратите внимание, что на самом деле 150 байт – это не «большой» блок. Обычно буфер имеет размер 4096 байт, если не изменить его с помощью `setvbuf`.)

На некоторых платформах вызов `ftell` выталкивает буфер, как побочный эффект, потому что это упрощает учет данных в библиотеке; библиотеки не любят, когда их ловят на лжи. (Вызов `fseek` также, скорее всего, вытолкнет буфер.) Однако есть платформы, в которых `ftell` и даже `fseek` никогда не выталкивают буфер. Чтобы гарантировать запись буфера потока `FILE` в соответствующий файл, используйте `fflush`. Продолжим предыдущий пример:

```
// Принудительно вытолкнуть буфер объекта FILE.
fflush(fp);

// Теперь fd и fp сообщают одинаковое состояние файла.
assert(lseek(fd, 0, SEEK_CUR) == 160);
```

С учетом всего вышесказанного, нашу простую программу из раздела «POSIX API» можно переписать с использованием `<stdio.h>`, как показано ниже:

```
#include <cassert>
#include <cstdio>
#include <string>

int main()
{
```

```
    FILE *fpw = fopen("myfile.txt", "w");
    FILE *fpr = fopen("myfile.txt", "r");
    if (fpw == nullptr || fpr == nullptr)
      return EXIT_FAILURE;

    fwrite("hello world", 1, 11, fpw);
    fseek(fpw, 6, SEEK_SET);
    fwrite("neighbor", 1, 8, fpw);
    fflush(fpw);

    std::string buffer(14, '\0');
    int b = fread(buffer.data(), 1, 14, fpr);
    assert(b == 14 && buffer == "hello neighbor");
    fclose(fpr);
    fclose(fpw);
}
```

На этом мы завершаем исследование возможностей буферизации в стандартной библиотеке `<stdio.h>` и переходим к поддержке форматирования в ней же.

Форматирование с помощью printf и snprintf

На этапе форматирования у нас изначально имеются высокоуровневые данные, которые требуется напечатать; например, нам может понадобиться напечатать количество настройщиков пианино в Чикаго, которое наша программа определила как 225. Напечатать (то есть вывести) *трехбайтную строку* `"225"` просто; мы уже решили эту задачу в предыдущих разделах. Задача же *форматирования* заключается в том, как из числа 225 (например, типа `int`) получить эту самую трехбайтную строку `"225"`.

При выводе чисел мы сталкиваемся со множеством проблем: в какой системе счисления должно быть представлено выводимое число? Если число отрицательное, мы должны добавить знак - перед ним, а если число положительное, нужно ли добавлять знак +? Должны ли мы использовать разделители групп разрядов, и если да, какой знак использовать в этом качестве – запятые, точки или пробелы? А как быть с десятичными точками? Сколько цифр печатать после десятичной точки? Или, может быть, мы должны использовать научную форму записи, и если да, сколько значащих разрядов выводить?

Вывод нечисловых данных тоже сопряжен со множеством вопросов. Должны ли мы выводить значение в поле фиксированной ширины, и если да, как его выравнивать – по левому краю, по правому или как-то иначе? (И какие символы использовать для заполнения пустых знакомест?) А как быть со значениями, не укладывающимися в отведенную ширину, – должны ли мы усечь значение слева или справа или, может быть, следует вывести значение целиком, невзирая на установленные ограничения?

Аналогично при чтении форматированных чисел (то есть выполняя парсинг) мы должны ответить практически на те же самые вопросы: можно ли

ожидать встретить разделители групп разрядов? Научную форму записи? Ведущие знаки +? В какой системе счисления записаны анализируемые числа? И для нечисловых данных: могут ли присутствовать ведущие пробелы? При чтении «строковых значений» какой еще знак может отмечать конец строки, кроме EOF?

Стандартный C API включает целое семейство функций форматирования с именами, оканчивающимися на printf, и соответствующее семейство функций парсинга с именами, оканчивающимися на scanf. Общей чертой всех функций в каждом семействе является список аргументов переменной длины и единственная «строка форматирования», предшествующая этому списку, которая отвечает на многие из вопросов, поставленных выше (но не на все), в отношении форматирования каждого аргумента, а также предоставляет «форму» для сообщения, в которую библиотека вставит отформатированные аргументы:

```
int tuners = 225;
const char *where = "Chicago";
printf("There are %d piano tuners in %s.\n", tuners, where);
```

Существуют также функции: fprintf(fp, "format", args...) – для вывода в произвольный поток (не обязательно stdout); snprintf(buf, n, "format", args...) – для вывода в буфер buf, о ней рассказывается чуть ниже; и соответствующее семейство функций vprintf, vfprintf и vsnprintf, которые могут пригодиться для создания своих printf-подобных функций. Как вы, уже вероятно, понимаете, полноценное обсуждение строк форматирования в стиле C выходит далеко за рамки этой книги. Тем не менее отмечу, что «язык строк форматирования» в стиле C распространился даже на языки, не являющиеся прямыми потомками C; например, на Python 2 можно написать такой код:

```
tuners = 225
where = "Chicago"
print "There are %d piano tuners in %s." % (tuners, where)
```

Однако между происходящим в C и в Python большая разница!

Самое большое отличие состоит в том, что Python – язык с динамической типизацией, поэтому, если записать "%s tuners" % (tuners), инструкция все равно выполнится правильно. В списках аргументов переменной длины в стиле C исходный тип type теряется; если указать спецификатор формата "%s" (который соответствует аргументу const char *) для аргумента типа int, вы в лучшем случае получите предупреждение компилятора, а в худшем – неопределенное поведение. То есть когда вы используете функции форматирования из <stdio.h>, строка формата выполняет двоякую роль: она определяет *не только формат* каждого значения, *но также его тип*, и если попытаться передать значение с типом, несовместимым со спецификатором формата, например использовать "%s" вместо "%d", в вашей программе появится ошибка. К счастью, большинство современных компиляторов своевременно обнаруживают такие

несоответствия, при условии что строка формата передается непосредственно в `printf` или функцию, отмеченную (нестандартным) атрибутом `format`, как будет показано в примере ниже. К сожалению, эта диагностика работает не всегда надежно, когда в дело вступают типы, зависящие от платформы; например, некоторые 64-разрядные компиляторы могут не обнаруживать попытку отформатировать значение типа `size_t` с помощью спецификатора формата `"%llu"`, даже притом, что переносимый спецификатор имеет вид `"%zu"`.

Другое отличие: функция `printf` в языке C фактически пишет непосредственно в стандартный поток вывода, `stdout`; форматирование данных совмещается с буферизацией выводимых байтов. В языке Python конструкция `"There are %d piano tuners in %s." % (tuners, where)` в действительности является *выражением* типа `str` (string – строка); собственно форматирование происходит непосредственно в нем, и создается законченное строковое значение, которое затем мы решили вывести в `stdout`.

Чтобы получить отформатированную строку с применением `<stdio.h>`, мы должны использовать `snprintf`:

```
char buf[13];
int needed = snprintf(
  buf, sizeof buf,
  "There are %d piano tuners in %s", tuners, where
);

assert(needed == 37);
assert(std::string_view(buf) == "There are 22");
```

Обратите внимание, что `snprintf` всегда добавляет нулевой символ в конец буфера, даже если при этом сообщение окажется неполным; и возвращает длину сообщения, которое она *могла бы* записать. Поэтому для получения форматированного сообщения с произвольной длиной часто сначала вызывается `snprintf` с пустым указателем на буфер, чтобы узнать длину будущего сообщения; а затем вызывается с буфером требуемого размера:

```
template<class... Args>
std::string format(const char *fmt, const Args&... args)
{
  int needed = snprintf(nullptr, 0, fmt, args...);
  std::string s(needed + 1, '\0');
  snprintf(s.data(), s.size(), fmt, args...);
  s.pop_back(); // удалить записанный '\0'
  return s;
}

void test()
{
  std::string s = format("There are %d piano tuners in %s", tuners, where);
  assert(s == "There are 225 piano tuners in Chicago");
}
```

Предыдущая реализация функции `format` использует шаблон функции со списком аргументов переменной длины, который страдает склонностью производить большое количество похожих копий кода. Эффективнее (в терминах времени компиляции и объема выполняемого кода) было создать единственную (нешаблонную) функцию в стиле языка C с переменным количеством аргументов и использовать `vsnprintf` для форматирования. К сожалению, более подробное обсуждение `va_list` и `vsnprintf` выходит далеко за рамки книги.

```
std::string better_format(const char *fmt, ...)
{
  va_list ap;
  va_start(ap, fmt);
  int needed = vsnprintf(nullptr, 0, fmt, ap);
  va_end(ap);
  std::string s(needed + 1, '\0');
  va_start(ap, fmt);
  vsnprintf(s.data(), s.size(), fmt, ap);
  va_end(ap);
  s.pop_back(); // удалить записанный '\0'
  return s;
}
```

Строки формата для `scanf` мы рассмотрим немного позже, в этой же главе. А полное описание `scanf` вы найдете на сайте cppreference.com или в книгах, посвященных стандартной библиотеке языка C.

Рассмотрев вкратце, как осуществляются буферизация и форматирование (по крайней мере, форматирование вывода) в `<stdio.h>`, перейдем к стандартному C++ `<iostream>` API.

Классическая иерархия потоков ввода/вывода

`<stdio.h>` API страдает, по меньшей мере, от трех проблем. Во-первых, функции форматирования не являются типобезопасными. Во-вторых, функции буферизации не совсем удачно разбиты на «функции буферизации в файлы» (`FILE *` и `fprintf`) и «функции буферизации в символьные буферы» (`snprintf`). (Ладно, ладно. Технически библиотека GNU C поддерживает `fopencookie` для конструирования `FILE *`, который выполняет буферизацию где угодно по вашему выбору; но это очень малопонятная и нестандартная функция.) В-третьих, нет простого способа расширить поддержку форматирования в пользовательских классах; я не могу передать в `printf` даже `std::string`, не говоря уже о `my::Widget`!

Когда C++ только разрабатывался в середине 1980-х, проектировщики осознали необходимость иметь типизированную, сборную и расширяемую библиотеку ввода/вывода. Так родилась библиотека, известная как «iostreams», или просто «потоки C++» (не путайте с потоками `<stdio.h>`, которые мы

только что закончили обсуждать). Фундаментальная архитектура библиотеки iostreams не изменилась с середины 1980-х, что очень отличает ее от всей остальной стандартной библиотеки, за исключением (это не каламбур) иерархии `std::exception`.

Библиотека C++ iostreams делится на две основные части: *потоки данных* (streams) – поддержку форматирования, – и `streambuf` – поддержку буферизации. Большинству программистов на C++ никогда не придется взаимодействовать с `streambuf`, только с потоками данных. Тем не менее давайте кратко познакомимся с понятием `streambuf`.

Механизм `streambuf` очень напоминает `FILE` в C API. Он сообщает программе, где находятся входные данные (в форме обычных байтов) и куда должны записываться выходные данные. Он также поддерживает буфер байтов для уменьшения обращений к соответствующим адресатам (таким как POSIX-функции `read` и `write`). Для поддержки разных реализаций `streambuf` с одним и тем же интерфейсом используется классический полиморфизм. Помните, как в начале главы я обещал, что мы еще встретимся с классическим полиморфизмом? Мы наконец подошли к этому!

`std::streambuf` (в действительности псевдоним для `std::basic_streambuf<char, char_traits<char>>`, но давайте не будем усложнять) – это базовый класс иерархии, который наследуют `std::filebuf` и `std::stringbuf`. Интерфейс `streambuf` содержит слишком много виртуальных методов, чтобы перечислить их все, поэтому отмечу лишь следующие: `sb.setbuf(buf, n)` (соответствует функции `setvbuf(fp, buf, _IO_FBF, n)`), `sb.overflow()` (соответствует функции `fflush(fp)`) и `sb.seekpos(offset, whence)` (соответствует функции `fseek(fp, offset, whence)`). Говоря о соответствии, я, конечно же, подразумеваю `std::filebuf`. Эти методы имеют в `std::stringbuf` иную реализацию (и обычно непереносимую).

Любой класс, производный от `streambuf`, должен также поддерживать некоторые элементарные операции для взаимодействий его с буфером (добавляющие и извлекающие байты). Эти элементарные операции предназначены не для обычных программистов, а для использования в объекте потока, который обертывает `streambuf` и реализует более дружественный высокоуровневый интерфейс.

Потоки данных в C++ инкапсулируют механизм `streambuf` и ограничивают множество операций с ним. Например, обратите внимание, что `streambuf` не имеет понятия «режим доступа»: вы можете добавлять байты в него («записывать») и извлекать байты («читать»). Однако если говорить о `streambuf`, завернутом в `std::ostream`, объект `ostream` экспортирует только метод `write` – он не имеет метода `read`.

На рис. 9.1 изображена диаграмма иерархии классов потоков и `streambuf` в C++17, как она определена в стандартных заголовках `<iostream>`, `<fstream>` и/или `<sstream>`. Базовые классы `streambuf`, `istream`, `ostream` и `iostream` являются «абстрагированными»: они не имеют чисто виртуальных методов и включают только одну переменную-член `streambuf*`, унаследованную из `ios`.

Чтобы не дать вам возможности по ошибке создать экземпляры этих «абстрагированных» типов, стандартная библиотека определяет их конструкторы как защищенные (`protected`). Напротив, классы с именами, содержащими `stringstream` и `fstream`, фактически включают экземпляры `stringbuf` и `filebuf` соответственно, конструкторы которых инициализируют унаследованный член `streambuf*`. Далее в этой главе, в разделе «Решение проблемы манипуляторов», вы увидите, как сконструировать объект `ostream`, в котором член `streambuf*` указывает на экземпляр `streambuf`, не принадлежащий `*this`:

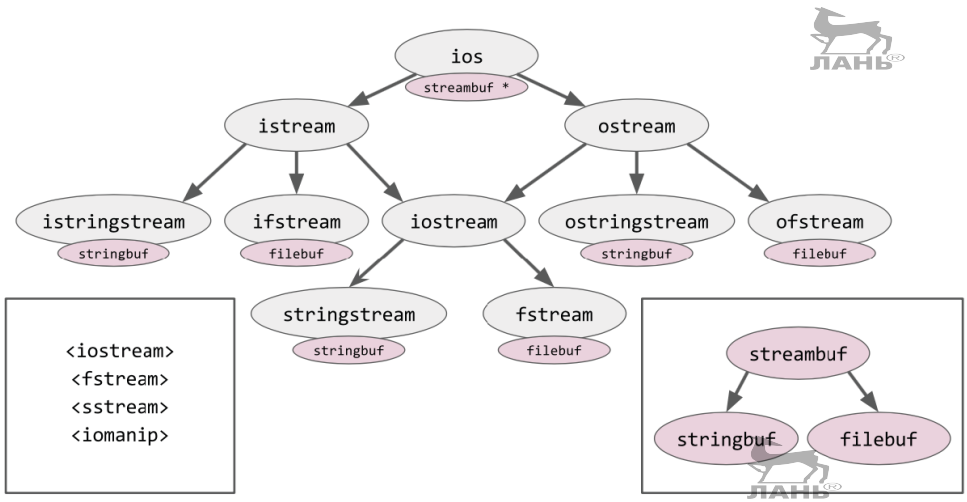


Рис. 9.1. Иерархия классов потоков и streambuf в C++17

Классы потоков данных экспортируют пеструю коллекцию методов, более или менее соответствующих функциям, которые мы уже дважды видели выше. В частности, класс `fstream` обертывает `filebuf`, и вместе они действуют подобно `FILE` из C API: `filebuf` имеет «курсор», которым можно управлять с помощью метода `seekp` класса `fstream`. (Имя `seekp` унаследовано от класса `ostream`. В `ifstream` похожий метод имеет имя `seekg`, где «g» в имени – от слова «get» (получить, прочитать), а буква «p» – от слова «put» (вставить, записать). В полноценном потоке `fstream` можно использовать любой из методов – `seekg` или `seekp` – в этом случае они являются синонимами. Как обычно, помните, что есть только один курсор, даже притом, что в iostreams API существует два метода для управления им!)

Конструктор `fstream` принимает битовую маску, состоящую из флагов `std::ios_base::in`, `out`, `app` (от англ. «append mode» – режим добавления в конец), `trunc`, `ate` и `binary`, объединенных по ИЛИ; однако, как мы видели на примере `fopen`, они лишь опосредованно связаны с флагами POSIX-функции `open`:

- `in`: эквивалент `fopen("r")`, или POSIX-флага `O_RDONLY`;

- `out`: эквивалент `fopen("w")`, или комбинации POSIX-флагов `O_WRONLY | O_CREAT | O_TRUNC` (обратите внимание на наличие флага `O_TRUNC`, даже притом, что флаг `trunc` не передается явно!);
- `in|out`: эквивалент `fopen("r+")`, или POSIX-флагов `O_RDWR | O_CREAT`;
- `in|out|trunc`: эквивалент `fopen("w+")`, или POSIX-флагов `O_RDWR | O_CREAT | O_TRUNC` (обратите внимание, что в данном случае синтаксис iostreams более последовательный, чем синтаксис `fopen`);
- `out|app`: эквивалент `fopen("a")`, или POSIX-флагов `O_WRONLY | O_CREAT | O_APPEND`;
- `in|out|app`: эквивалент `fopen("a+")`, или POSIX-флагов `O_RDWR | O_CREAT | O_APPEND`.

Добавление флага `binary` в битовую маску сродни добавлению символа `"b"` в строку режима функции `fopen`. Флаг `ate` сообщает потоку данных, что изменение текущей позиции будет производиться относительно конца файла, даже если файл открыт в режиме `O_APPEND`.

Если передать неподдерживаемый набор флагов, например `app|trunc`, объект потока все равно будет создан, но помещен в «нерабочее» состояние, которое мы вскоре обсудим. В общем случае следует проектировать конструкторы собственных классов так, чтобы они сообщали об ошибках посредством исключений. Здесь это правило нарушено, отчасти потому, что данная иерархия классов проектировалась почти сорок лет назад, а отчасти потому, что нам нужен некоторый механизм «сбоев», чтобы обрабатывать весьма вероятные ситуации, когда требуемый файл не может быть открыт (например, потому что не существует).

Совместив все вместе, нашу простую программу из раздела «POSIX API» можно переписать с использованием `<fstream>` API, как показано ниже:

```
#include <cassert>
#include <fstream>
#include <string>

int main()
{
  std::fstream fsw("myfile.txt", std::ios_base::out);
  std::fstream fsr("myfile.txt", std::ios_base::in);
  if (fsw.fail() || fsr.fail())
    return EXIT_FAILURE;

  fsw.write("hello world", 11);
  fsw.seekp(6, std::ios_base::beg);
  fsw.write("neighbor", 8);
  fsw.flush();

  std::string buffer(14, '\0');
  fsr.read(buffer.data(), 14);
```

```
    assert(fsr.gcount() == 14 && buffer == "hello neighbor");
}
```

В предыдущем примере есть одна странность: `fsr.read(buffer.data(), 14)` не возвращает ничего, что могло бы подсказать, сколько байтов было прочитано! Вместо этого число прочитанных байтов сохраняется в переменной-члене и его можно получить отдельным вызовом функции `fsr.gcount()`. Аналогично метод `write` не дает возможности узнать, сколько байтов было фактически записано. Это может выглядеть как проблема, но вообще, если поток данных сталкивается с ошибкой в процессе чтения или записи, эта ошибка часто оказывается «неисправимой» из-за неизвестного количества байтов, *фактически* прочитанных или записанных в файловый дескриптор, и из-за наличия нескольких уровней буферизации между прикладной программой и аппаратурой. Столкнувшись с ошибкой чтения или записи, предпочтительнее вообще отказаться от попытки понять, в каком состоянии оказался поток, – *кроме* особого случая, когда при чтении достигнут «конец файла». Если предполагалось прочитать 100 байт и в процессе чтения достигнут «конец файла», определенно есть смысл спросить: «Сколько байтов было прочитано?» Однако если предполагалось записать 100 байт и в процессе была получена сетевая ошибка или ошибка диска, бессмысленно пытаться узнать, «сколько байтов было благополучно записано», потому что мы не сможем с полной уверенностью утверждать, что «благополучно записанные» байты действительно достигли места назначения.

Если мы затребовали 100 байт, а получили 99 (или меньше), из-за того, что достигнут конец файла, тогда не только `fs.gcount()` вернет число меньше 100, но также состояние объекта потока получит значение – *признак конца файла*. Определить текущее состояние потока можно с помощью функций `fs.good()` (все замечательно?), `fs.bad()` (поток столкнулся с неисправимой ошибкой?), `fs.eof()` (последняя операция чтения достигла конца файла?) и `fs.fail()` (последняя операция привела поток в «нерабочее» состояние по какой-то причине?). Обратите внимание, что `fs.good()` не является инверсией `fs.bad()`; поток может, например, находиться в состоянии `eof`, когда выполняется условие `!good() && !bad()`.

К настоящему моменту мы увидели простейший способ буферизации ввода и вывода с использованием потоков `fstream`. Однако если бы потоки C++ поддерживали только такой способ использования, мы с равным успехом могли бы ограничиться `FILE *` или даже POSIX API. Новые и (вероятно) улучшенные возможности потоков C++ заключаются в особенностях работы с *форматированием*.

Потоки данных и манипуляторы

Как вы наверняка помните, в функции `printf` исходные типы аргументов теряются, поэтому строка формата должна выполнять двойную функцию, кодировать не только *формат* вывода каждого значения, но также его *тип*.

В iostreams этот недостаток отсутствует. Вот как выглядит форматирование с iostreams:

```
int tuners = 225;
const char *where = "Chicago";
std::cout << "There are " << tuners << " piano tuners in " << where << "\n";
```

Здесь `std::cout` – это глобальная переменная типа `ostream`, соответствующая стандартному потоку вывода `stdout` или дескриптору файла 1 в POSIX. Существует также переменная `std::cerr`, соответствующая небуферизованному потоку `stderr` или дескриптору файла 2 в POSIX; `std::clog`, также соответствующая дескриптору файла 2, но с полной буферизацией; и `std::cin` – глобальная переменная типа `istream`, соответствующая стандартному потоку вывода `stdin` или дескриптору файла 0 в POSIX.

Стандартный класс `ostream` (который, опять же, в действительности является `basic_ostream<char, char_traits<char>>`, но давайте не будем акцентировать свое внимание на этом) имеет много, очень много перегруженных нечленов `operator<<`. Например, вот простейший перегруженный `operator<<`:

```
namespace std {
  ostream& operator<< (ostream& os, const string& s)
  {
    os.write(s.data(), s.size());
    return os;
  }
} // namespace std
```

Так как эта функция возвращает ссылку на тот же объект `os`, который она получила, мы можем составлять цепочки из операторов <<, как показано в предыдущем примере. Это позволяет форматировать сложные сообщения.

К сожалению, нашего простого `operator<<(ostream&, const string&)` недостаточно для решения различных задач форматирования, описанных в разделе «Форматирование с помощью printf и snprintf». Представьте, что нам нужно вывести строку в поле шириной 7 символов с выравниванием по левому краю; как сделать это? Синтаксис `operator<<` не предусматривает передачу дополнительных «параметров форматирования». То есть мы просто не можем выполнять сложное форматирование, если не определить параметры форматирования либо слева от оператора << (в самом объекте `ostream`), либо справа (в форматируемом объекте). Стандартная библиотека использует комбинацию двух подходов. Вообще говоря, функциональность, созданная в 1980-х и 1990-х, предусматривала параметры форматирования в самом объекте `ostream`; и все, что было добавлено позднее, – вследствие невозможности добавить в `ostream` переменные-члены, не нарушив двоичную совместимость – пришлось реализовывать справа от оператора <<. Давайте посмотрим пример выравнивания в поле на основе подходов 1980-х. Вот немного более полная версия нашего `operator<<` для `std::string`:

```
void pad(std::ostream& os, size_t from, size_t to)
{
  char ch = os.fill();
  for (auto i = from; i < to; ++i) {
    os.write(&ch, 1);
  }
}

std::ostream& operator<< (std::ostream& os, const std::string& s)
{
  auto column_width = os.width();
  auto padding = os.flags() & std::ios_base::adjustfield;

  if (padding == std::ios_base::right) {
    pad(os, s.size(), column_width);
  }
  os.write(s.data(), s.size());
  if (padding == std::ios_base::left) {
    pad(os, s.size(), column_width);
  }
  os.width(0); // сбросить "ширину поля" в 0
  return os;
}
```

Здесь `os.width()`, `os.flags()` и `os.fill()` – это встроенные члены класса `std::ostream`. Существует также член `os.precision()` для форматирования вещественных чисел, и с помощью `os.flags()` можно для некоторых числовых типов задать систему счисления по основанию 10, 16 или 8. Ширину поля в потоке можно установить вызовом `os.width(n)`, однако было бы слишком утомительно (и глупо!), если бы перед каждой операцией вывода мы были вынуждены выполнять настройки `std::cout.width(10)`, `std::cout.setfill('.')` и другие. Поэтому библиотека iostreams включает стандартные *манипуляторы потоком*, позволяющие получить тот же эффект, что дают эти функции-члены, но более «потокообразным» способом. Большинство манипуляторов определено в стандартном заголовке `<iomanip>`, а не в `<iostream>`. Например, вот манипулятор, управляющий шириной поля в потоке:

```
struct WidthSetter { int n; };

auto& operator<< (std::ostream& os, WidthSetter w)
{
  os.width(w.n);
  return os;
}

auto setw(int n) { return WidthSetter{n}; }
```

А вот еще два стандартных манипулятора, один из которых выглядит очень знакомо. Манипулятор `std::endl` добавляет символ перевода строки в поток и затем выталкивает его:

```
using Manip = std::ostream& (*)(std::ostream&);

auto& operator<< (std::ostream& os, Manip f) {
  return f(os);
}

std::ostream& flush(std::ostream& os) {
  return os.flush();
}

std::ostream& endl(std::ostream& os) {
  return os << '\n' << flush;
}
```

Также имеются манипуляторы `std::setw` и родственные ему `std::left`, `std::right`, `std::hex`, `std::dec`, `std::oct`, `std::setfill`, `std::precision` и все остальные, позволяющие писать код iostreams, который выглядит очень естественно, хотя и избыточно подробно. Сравните следующие фрагменты с `<stdio.h>` и `<iostream>`:

```
printf("%-10s.%6x\n", where, tuners);
  // "Chicago . e1"

std::cout << std::setw(8) << std::left << where << "."
          << std::setw(4) << std::right << std::hex
          << tuners << "\n";
  // "Chicago . e1"
```

Имейте в виду, что, используя эти манипуляторы, мы неизбежно оказываем влияние на состояние самого объекта потока; это влияние может сохраняться дольше, чем текущий вывод. Например, предыдущий пример можно продолжить следующим кодом:

```
printf("%d\n", 42);      // "42"
std::cout << 42 << "\n"; // "2a" -- ой!
```

Манипулятор `std::hex` из предыдущего примера настраивает поток на «вывод чисел в шестнадцатеричном формате», и по завершении вывода поток не возвращается в десятичный режим «по умолчанию». То есть с этого момента все последующие числа программа будет выводить в шестнадцатеричном формате! Это серьезный недостаток библиотеки iostreams (и императивного программирования в целом).

Потоки данных и обертки

Параметры, поддерживаемые классом `std::ios_base` (`left`, `right`, `hex`, `width`, `precision` и т. д.), представляют ограниченное множество, которое было определено в середине 1980-х и с тех пор оставалось в неприкосновенности. Так как каждый манипулятор изменяет свой параметр в состоянии потока,

множество манипуляторов также остается, по сути, ограниченным. В настоящее время принято организовывать форматирование конкретных данных, заворачивая их в *обертки*. Например, представьте, что у нас имеется обобщенный алгоритм для заключения значений в кавычки:

```
template<class InputIt, class OutputIt>
OutputIt do_quote(InputIt begin, InputIt end, OutputIt dest)
{
  *dest++ = '"';
  while (begin != end) {
    auto ch = *begin++;
    if (ch == '"') {
      *dest++ = '\\';
    }
    *dest++ = ch;
  }
  *dest++ = '"';
  return dest;
}
```

(Этот алгоритм не является частью стандартной библиотеки.) Имея этот алгоритм, легко можно сконструировать класс обертки, оператор `operator<<` которого мог бы вызывать следующий алгоритм:

```
struct quoted {
  std::string_view m_view;
  quoted(const char *s) : m_view(s) {}
  quoted(const std::string& s) : m_view(s) {}
};

std::ostream& operator<< (std::ostream& os, const quoted& q)
{
  do_quote(
    q.m_view.begin(),
    q.m_view.end(),
    std::ostreambuf_iterator<char>(os)
  );
  return os;
}
```

(Тип `std::ostreambuf_iterator<char>` является частью стандартной библиотеки и определяется в заголовке `<iterator>`. Далее в этой главе мы познакомимся с дружественным ему итератором `istream_iterator`.) Далее, имея класс обертки, можно писать весьма выразительный код для вывода значений в кавычках:

```
std::cout << quoted("I said \"hello\".");
```

Обертка, которую мы только что придумали, преднамеренно сделана похожей на функцию-обертку `std::quoted`, которая объявляется в стандартном

заголовке <iomanip>. Главное отличие: std::quoted не использует алгоритмов с итераторами; она конструирует весь вывод в локальной переменной std::string и затем выполняет os << str, чтобы вывести результата одним махом. Это означает, что std::quoted не поддерживает возможности смены диспетчера памяти (см. главу 8 «Диспетчеры памяти») и потому не подходит для окружений, где выделение памяти в куче невозможно. Несмотря на мелкие шероховатости, в целом идея использования функции-обертки или класса для настройки форматирования значения данных выглядит довольно привлекательно. В таких библиотеках, как Boost.Format, она получила сильное развитие и допускает возможность использовать такой синтаксис:

```
std::cout << boost::format("There are %d piano tuners in %s.") % tuners % where
<< std::endl;
```

Обертки, определяющие автономные операции форматирования, предпочтительнее, чем манипуляторы, которые изменяют состояние потока. В примере выше мы видели, как манипулятор std::hex может сказываться на всех последующих операциях вывода чисел. Теперь рассмотрим два решения этой проблемы – и две новые проблемы, возникающие на ее месте!

Решение проблемы манипуляторов

Нашу «проблему манипулятора std::hex» можно решить заключением сложных операций вывода в операции сохранения и восстановления состояния потока ostream или созданием совершенно нового экземпляра ostream перед каждым выводом. Пример первого решения:

```
void test() {
  std::ios old_state(nullptr);
  old_state.copyfmt(std::cout);
    std::cout << std::hex << 225; // "e1"
  std::cout.copyfmt(old_state);

  std::cout << 42; // "42"
}
```

И второго:

```
void test() {
  std::ostream os(std::cout.rdbuf());
  os << std::hex << 225; // "e1"

  std::cout << 42; // "42"
}
```

Обратите внимание, насколько удобным оказалось отделение идеи «streambuf» от идеи «поток данных» в библиотеке iostreams; в предыдущем примере мы легко отделили все поля, связанные с форматированием, от потока, из-

влекая только его streambuf: std::cout.rdbuf() и накладывая совершенно новый поток (с собственными полями для форматирования) поверх того же streambuf.

Однако форматирование в iostreams имеет еще один существенный недостаток. Каждая часть сообщения выводится «немедленно», по достижении соответствующего operator<<, или, если хотите, вычисление каждой части сообщения «откладывается» до достижения соответствующего operator<<, то есть следующий фрагмент кода:

```
void test() {
  try {
    std::cout << "There are "
              << computation_that_may_throw()
              << "piano tuners here.\n";
  } catch (...) {
    std::cout << "An exception was thrown";
  }
}
```

выведет: There are An exception was thrown.

Кроме того, форматирование в iostreams *сильно* затрудняет интернационализацию («i18n»), потому что «форма» всего сообщения отсутствует в исходном коде. Вместо единственного строкового литерала "There are %d piano tuners here.\n", представляющего полную фразу для вывода, который человек мог бы перевести на другой язык и сохранить во внешнем файле с переведенными сообщениями у нас имеются два фрагмента предложения: "There are " и "piano tuners here.\n", – ни один из которых нельзя перевести по отдельности.

По всем этим причинам я настоятельно не рекомендую использовать iostreams как *основу*. <stdio.h> и сторонние библиотеки форматирования, такие как fmt, в этом отношении выглядят предпочтительнее. Также можно использовать Boost.Format, хотя эта библиотека существенно увеличивает время компиляции и имеет посредственную производительность в сравнении с двумя предыдущими вариантами. Если вы заметите, что вводите <<, std::hex или os.rdbuf() чаще, чем один-два раза в неделю, значит, что-то вы делаете неправильно.

И вместе с тем библиотека iostreams имеет ряд удобных и даже полезных особенностей! Рассмотрим одну из них.

Форматирование с ostringstream

До сих пор мы говорили в основном о классе fstream, который приблизительно соответствует функциям форматирования fprintf и vfprintf из C API. Однако существует также класс ostringstream, который соответствует функциям snprintf и vsnprintf.

Класс ostringstream, так же как ostream, поддерживает все обычные функции operator<<; однако запись выполняется не в дескриптор файла, а в сим-

вольный буфер изменяемого размера – на практике `std::string`! С помощью метода `oss.str()` вы можете получить копию этого буфера и использовать ее для своих нужд. Такая организация делает возможной следующую идиому «преобразования в строку» объекта произвольного типа `T`:

```
template<class T>
std::string to_string(T&& t)
{
  std::ostringstream oss;
  oss << std::forward<T>(t);
  return oss.str();
}
```

В C++17 можно даже реализовать версию `to_string` с произвольным количеством аргументов:

```
template<class... Ts>
std::string to_string(Ts&&... ts)
{
  std::ostringstream oss;
  (oss << ... << std::forward<Ts>(ts));
  return oss.str();
}
```

С этой версией вызов, такой как `to_string(a, "  ", b)` или `to_string(std::hex, 42)`, будет иметь соответствующую семантику.

Примечание о региональных настройках

Существует еще одна ловушка, которой следует остерегаться при использовании `printf` или `ostream` для форматирования (или парсинга) строк. Эта ловушка – *локали*. Полное обсуждение локалей выходит далеко за рамки этой книги, но, если говорить кратко, *локаль* (или *региональные настройки*) – это «подмножество окружения пользователя, зависящее от естественного языка и культурных особенностей». Региональные настройки, доступные программно через службы операционной системы, позволяют программе корректировать свое поведение в зависимости от настроек, выбранных пользователем. Например, с их помощью можно определить, является ли «á» алфавитным символом, с какого дня начинается неделя – с воскресенья или с понедельника, какой формат использовать для вывода дат – «23-01-2017» или «01-23-2017» и как печатать вещественные числа – как «1234.56» или «1.234,56». Программист из XXI века, прочитав эти строки, может воскликнуть: «Это же безумие! Вы хотите сказать, что ничего из этого не определяется стандартом? Полагаю, что такое положение вещей неизбежно будет приводить к тонким и болезненным ошибкам!» И он будет прав!

```
std::setlocale(LC_ALL, "C.UTF-8");
```

```
std::locale::global(std::locale("C.UTF-8"));

auto json = to_string('[', 3.14, ']');
assert(json == "[3.14]"); // Успех!

std::setlocale(LC_ALL, "en_DK.UTF-8");
std::locale::global(std::locale("en_DK.UTF-8"));

json = to_string('[', 3.14, ']');
assert(json == "[3,14]"); // Ошибка без прерывания программы!
```

Выбрав глобальную локаль `"en_DK.UTF-8"`, мы получили неправильный вывод в JSON. Горе несчастному пользователю, который попытается запустить веб-сервер или базу данных с любой локалью, отличной от `"C.UTF-8"`!

При программировании, помимо потерь из-за соблюдения правил, диктуемых региональными настройками, мы должны также бороться с потерями производительности. Обратите внимание, что «текущая локаль» является глобальной переменной, а это означает, что каждое обращение к ней должно оформляться как атомарная операция или, хуже того, защищаться глобальным мьютексом. И каждый вызов `snprintf` или `operator<<(ostream&, double)` должен использовать текущую локаль. Это влечет жуткие потери производительности и, в некоторых случаях, может превратиться в узкое место в многопоточном коде.

Как прикладной программист приложений со сложностью выше определенного уровня вы должны взять в привычку писать `std::locale::global(std::locale("C"))` в первой строке функции `main()`. (Если написать просто `setlocale(LC_ALL, "C")`, как это делается в программах на C, библиотека `<stdio.h>` будет работать правильно, но это никак не повлияет на использование локали библиотекой `<iostream>`. Иначе говоря, настройка «глобальной локали» в библиотеке C++ также изменяет «глобальную локаль» в библиотеке C, но не наоборот.)

Если вместо локали `"C"` вы решите использовать `"C.UTF-8"`, имейте в виду, что имя `"C.UTF-8"` было утверждено только в 2015 году и может не поддерживаться на старых системах. Фактически доступность любой локали, кроме `"C"`, зависит от того, как пользователь настроил систему. В этом отношении локали подобны часовым поясам: на всех платформах в мире гарантируется наличие только одной локали и одного часового пояса, и именно их вы должны использовать.

У вас как у разработчика сторонних библиотек на выбор есть два пути. Самый простой: предположить, что ваша библиотека всегда будет использоваться только в приложениях, которые устанавливают глобальную локаль `"C"`, и по-

тому у вас нет причин для беспокойства; сделать шаг вперед и взять за основу для вывода snprintf и operator<<. (Однако это не решает проблемы производительности при использовании локали. Вы все еще должны использовать глобальный мьютекс, расходуя драгоценные такты процессора.) Более сложный путь, потому что требует неукоснительного следования всем правилам: вообще избегать функций форматирования, которые обращаются к региональным настройкам. Этот путь стал осуществимым только с появлением в библиотеке C++17 некоторых новейших особенностей, с которыми мы сейчас познакомимся.

Преобразование чисел в строки

Возьмем за основу следующие объявления:

```
std::ostringstream oss;
std::string str;
char buffer[100];
int intvalue = 42;
float floatvalue = 3.14;
std::to_chars_result r;
```


Для преобразования целого числа intvalue в строку цифр библиотека C++17 предлагает следующие варианты:

```
snprintf(buffer, sizeof buffer, "%d", intvalue);
  // доступна в <stdio.h>
  // не зависит от локали (%d действует без учета локали)
  // не использует динамическую память
  // поддерживает системы счисления с основанием 8, 10 и 16

oss << intvalue;
str = oss.str();
  // доступна в <sstream>
  // зависит от локали (может вставлять разделители групп разрядов)
  // использует динамическую память; поддерживает диспетчеры памяти
  // поддерживает системы счисления с основанием 8, 10 и 16

str = std::to_string(intvalue);
  // доступна в <string> начиная с версии C++11
  // не зависит от локали (эквивалент %d)
  // использует динамическую память; НЕ поддерживает диспетчеры памяти
  // поддерживает только систему счисления с основанием 10

r = std::to_chars(buffer, std::end(buffer), intvalue, 10);
*r.ptr = '\0';
  // доступна в <charconv> начиная с версии C++17
  // не зависит от локали
  // не использует динамическую память
  // поддерживает системы счисления с основанием от 2 до 36
```

Все четыре альтернативы имеют свои преимущества. Главное преимущество std::to_string в удобстве конструирования больших сообщений в высокоуровневом коде:

```
std::string response =
  "Content-Length: " + std::to_string(body.size()) + "\r\n" +
  "\r\n" + body;
```

Главное преимущество std::to_chars – в независимости от локали и простоте использования в низкоуровневом коде:

```
char *write_string(char *p, char *end, const char *from)
{
  while (p != end && *from != '\0') *p++ = *from++;
  return p;
}

char *write_response_headers(char *p, char *end, std::string body)
{
  p = write_string(p, end, "Content-Length: ");
  p = std::to_chars(p, end, body.size(), 10).ptr;
  p = write_string(p, end, "\r\n\r\n");
  return p;
}
```

Главный недостаток std::to_chars – это слишком новая функция, появившаяся только в C++17; на момент написания этих строк заголовок <charconv> отсутствовал во всех основных реализациях стандартной библиотеки.

Для преобразования вещественного числа floatvalue в строку цифр библиотека C++17 предлагает следующие варианты:

```
snprintf(buffer, sizeof buffer, "%.6e", floatvalue);
snprintf(buffer, sizeof buffer, "%.6f", floatvalue);
snprintf(buffer, sizeof buffer, "%.6g", floatvalue);
  // доступна в <stdio.h>
  // зависит от локали (десятичная точка)
  // не использует динамическую память

oss << floatvalue;
str = oss.str();
  // доступна в <sstream>
  // зависит от локали (десятичная точка)
  // использует динамическую память; поддерживает диспетчеры памяти

str = std::to_string(floatvalue);
  // доступна в <string> начиная с версии C++11
  // зависит от локали (эквивалент %f)
  // использует динамическую память; НЕ поддерживает диспетчеры памяти
  // не позволяет настроить формат

r = std::to_chars(buffer, std::end(buffer), floatvalue,
```

```
                    std::chars_format::scientific, 6);
r = std::to_chars(buffer, std::end(buffer), floatvalue,
                    std::chars_format::fixed, 6);
r = std::to_chars(buffer, std::end(buffer), floatvalue,
                    std::chars_format::general, 6);
*r.ptr = '\0';
  // доступна в <charconv> начиная с версии C++17
  // не зависит от локали
  // не использует динамическую память
```

Примечательно, что все методы вывода вещественных чисел, кроме `std::to_string`, дают возможность настроить форматирование; и все методы, кроме `std::to_chars`, зависят от локали и потому усложняют разработку переносимого кода. Кроме `float`, все эти методы поддерживают также типы `double` и `long double`. Во всех случаях этим методам свойственны достоинства и недостатки, присущие методам форматирования целых чисел.

Преобразование строк в числа

Обратной форматированию чисел является задача *парсинга* чисел из ввода пользователя. Парсинг, – по своей природе намного более сложная и тонкая задача, потому что должна учитывать вероятность ошибки. Любое число можно превратить в строку цифр, но не всякую строку (и даже строку цифр) можно превратить в число. Поэтому любая функция, выполняющая парсинг чисел, должна предусматривать возможность обработки строк, не являющихся допустимыми представлениями чисел.

Возьмем за основу следующие объявления:

```
std::istringstream iss;
std::string str = "42";
char buffer[] = "42";
int intvalue;
float floatvalue;
int rc;
char *endptr;
size_t endidx;
std::from_chars_result r;
```

Для преобразования строки в `buffer` или `str` в целое число `intvalue` C++17 предлагает следующие варианты:

```
intvalue = strtol(buffer, &endptr, 10);
  // возможно переполнение
  // устанавливает глобальную переменную "errno" для большинства ошибок
  // устанавливает endptr==buffer, когда не может выполнить парсинг
  // доступна в <stdlib.h>
  // зависит от локали, теоретически
  // не использует динамическую память
```

```
  // поддерживает системы счисления с основанием 0 и от 2 до 36
  // всегда пропускает начальные пробелы
  // пропускает начальную пару символов 0x для системы счисления с основанием 16
  // распознает регистр символов

rc = sscanf(buffer, "%d", &intvalue);
  // завершается с ошибкой при обнаружении переполнения
  // возвращает 0 (вместо 1), когда не может выполнить парсинг
  // доступна в <stdio.h>
  // зависит от локали (эквивалент strtol)
  // не использует динамическую память
  // поддерживает системы счисления с основанием 0, 8, 10 и 16
  // всегда пропускает начальные пробелы
  // пропускает начальную пару символов 0x для системы счисления с основанием 16
  // распознает регистр символов

intvalue = std::stoi(str, &endidx, 10);
  // завершается с ошибкой при обнаружении переполнения
  // доступна в <string> начиная с C++11
  // зависит от локали (эквивалент strtol)
  // НЕ поддерживает диспетчеры памяти
  // поддерживает системы счисления с основанием 0 и от 2 до 36
  // всегда пропускает начальные пробелы
  // пропускает начальную пару символов 0x для системы счисления с основанием 16
  // распознает регистр символов

iss.str("42");
iss >> intvalue;
  // возможно переполнение
  // по любой ошибке устанавливает iss.fail()
  // доступна в <sstream>
  // зависит от локали
  // использует динамическую память; поддерживает диспетчеры памяти
  // поддерживает системы счисления с основанием 8, 10 и 16
  // пропускает начальную пару символов 0x для системы счисления с основанием 16
  // по умолчанию пропускает пробелы

r = std::from_chars(buffer, buffer + 2, intvalue, 10);
  // по любой ошибке устанавливает r.ec != 0
  // доступна в <charconv> начиная с C++17
  // не зависит от локали
  // не использует динамическую память
  // поддерживает системы счисления с основанием от 2 до 36
  // всегда пропускает начальные пробелы
  // распознает регистр символов
```

Методов парсинга оказалось больше, чем методов форматирования, описанных в предыдущем разделе; это объясняется тем, что только одна стандартная библиотека C предлагает три разных метода: `atoi`, самый старый и единственный, поведение которого не определено в отношении недопустимой исходной строки, поэтому его лучше не использовать; `strtol`, стандартная за-

мена для atoi, сообщающая об ошибке переполнения через глобальную переменную errno (из-за чего не подходит для использования в многопоточном или высокопроизводительном коде); и sscanf – функция из того же семейства, что и snprintf.

std::stoi – очень хорошая замена для atoi в сценариях однократного парсинга и очень плохая для высокопроизводительного кода. Эта функция прекрасно справляется с определением ошибок – вызов std::stoi("2147483648") возбудит исключение std::out_of_range, а вызов std::stoi("abc") возбудит исключение std::invalid_argument. (И хотя std::stoi("42abc") вернет 42 без ошибки, вызов std::stoi("42abc", &endidx) запишет в endidx число 2 вместо 5, указывая на вероятную ошибку.) Главный недостаток std::stoi – она работает только со строками типа std::string – нет перегруженной версии std::stoi для string_view, std::pmr::string и даже для const char *!

std::from_chars – наиболее современный и производительный инструмент для парсинга целых чисел. Главное достоинство, отличающее эту функцию от остальных конкурентов: from_chars не требует, чтобы входной буфер завершался нулевым символом, – она принимает пару указателей begin и end, определяющих диапазон символов для парсинга, и никогда не читает символы, следующие за end. Но имеет неприятные ограничения – например, не пропускает пробельные символы и не распознает шестнадцатеричные цифры в верхнем регистре. Идиома проверки r.ec на наличие ошибки показана в начале главы 12 «Файловая система».

Для функций strtol, sscanf и stoi в примерах выше указывается, что они поддерживают «систему счисления с основанием 0». Этот особый синтаксис, когда библиотеке передается основание системы счисления 0 (или, в случае с sscanf, спецификатор формата "%i"), сообщает, что парсинг строки должен производиться, как если бы она содержала литерал целого числа на языке C: 0123 будет интерпретироваться как восьмеричное представление десятичного числа 83, 0x123 будет интерпретироваться как шестнадцатеричное представление десятичного числа 291, и 019 будет интерпретироваться как восьмеричное представление целого числа 1, где символ 9 будет проигнорирован, потому что не является восьмеричной цифрой. «Основание 0» никогда не следует использовать в компьютерных программах, и from_chars благоразумно отвергает его.

Для преобразования строки в вещественное число floatvalue C++17 предлагает следующие варианты:

```
floatvalue = strtof(buffer, &endptr);
  // возможно переполнение
  // устанавливает глобальную переменную "errno" для большинства ошибок
  // устанавливает endptr==buffer, когда не может выполнить парсинг
  // доступна в <stdlib.h>
  // зависит от локали
  // не использует динамическую память
  // поддерживает системы счисления с основанием 10 и 16,
```

```
  //   определяются автоматически
  // всегда пропускает начальные пробелы

rc = sscanf(buffer, "%f", &floatvalue);
  // завершается с ошибкой при обнаружении переполнения
  // возвращает 0 (вместо 1), когда не может выполнить парсинг
  // доступна в <stdio.h>
  // зависит от локали (эквивалент strtof)
  // не использует динамическую память
  // поддерживает системы счисления с основанием 10 и 16,
  //   определяются автоматически
  // всегда пропускает начальные пробелы

floatvalue = std::stof(str, &endidx);
  // завершается с ошибкой при обнаружении переполнения
  // доступна в <string> начиная с C++11
  // зависит от локали (эквивалент strtof)
  // НЕ поддерживает диспетчеры памяти
  // поддерживает системы счисления с основанием 10 и 16,
  //   определяются автоматически
  // всегда пропускает начальные пробелы

iss.str("3.14");
iss >> floatvalue;
  // возможно переполнение
  // по любой ошибке устанавливает iss.fail()
  // доступна в <sstream>
  // зависит от локали
  // использует динамическую память; поддерживает диспетчеры памяти
  // поддерживает системы счисления с основанием 10 и 16,
  //   определяются автоматически
  // пропускает начальные пробелы по умолчанию
  // непереносимое поведение в отношении завершающего текста

r = std::from_chars(buffer, buffer + 2, floatvalue,
std::chars_format::general);
  // по любой ошибке устанавливает r.ec != 0
  // доступна в <charconv> начиная с C++17
  // не зависит от локали
  // не использует динамическую память
  // поддерживает системы счисления с основанием 10 и 16,
  //   определяются автоматически
  // всегда пропускает начальные пробелы
```

Все эти функции – даже `std::from_chars` – распознают строки `"Infinity"` и `"Nan"` (без учета регистра символов), а также распознают «шестнадцатеричные вещественные» числа, например строка `"0x1.c"` будет преобразована в вещественное число 1.75. Все функции, кроме `std::from_chars`, учитывают региональные настройки и потому усложняют разработку переносимого кода. Если проблемы с локалью при парсинге целых чисел носят в основном теоретиче-

ский характер, то разнообразие локалей, где точка (.) не является десятичным разделителем, легко может стать источником проблем при использовании `std::stof` и `std::stod`:

```
std::setlocale(LC_ALL, "C.UTF-8");
assert(std::stod("3.14") == 3.14); // Успех!
std::setlocale(LC_ALL, "en_DK.UTF-8");
assert(std::stod("3.14") == 3.00); // Ошибка без прерывания программы!
```

Обратите внимание на примечание «непереносимое поведение в отношении завершающего текста» в примере использования `istringstream`. Разные производители библиотеки по-разному поступают в этой ситуации, и не всегда очевидно, какое поведение можно назвать «правильным»:

```
double d = 17;
std::istringstream iss("42abc");
iss >> d;
if (iss.good() && d == 42) {
  puts("Your library vendor is libstdc++");
} else if (iss.fail() && d == 0) {
  puts("Your library vendor is libc++");
}
```

Из-за этих проблем с переносимостью, которые могут приводить к трудноуловимым ошибкам, я советую не использовать `istringstream` для парсинга ввода, даже притом, что в некоторых ситуациях `ostringstream` является наиболее подходящим вариантом для форматирования вывода.

Еще одно хорошее правило: отделяйте проверку ввода (разделение на лексемы) от парсинга. Если есть возможность заранее убедиться, что строка содержит только цифры или соответствует регулярному выражению, описывающему вещественное число, тогда вам проще будет выбрать метод парсинга, обнаруживающий переполнение и/или завершающий текст; например, `std::stof` или `std::from_chars`. Подробнее о выделении лексем с помощью регулярных выражений рассказывается в главе 10 «Регулярные выражения».

Чтение по одной строке или по одному слову

Чтение по одной строке из стандартного ввода – распространенная задача, и многие языки сценариев поддерживают для этого однострочные инструкции liner. Например, на Python:

```
for line in sys.stdin:
    # сохраняет завершающие символы перевода строки
    process(line)
и на Perl:
while (<>) {
  # сохраняет завершающие символы перевода строки
  process($_);
}
```

На C++ эта задача решается почти так же просто. Обратите внимание, что функция `std::getline` в C++, в отличие от других языков, удаляет завершающие символы перевода строки (если имеются) из прочитанных строк:

```
std::string line;
while (std::getline(std::cin, line)) {
  // автоматически удалит завершающие символы перевода строки
  process(line);
}
```

В каждом из этих случаев весь ввод никогда не окажется в памяти целиком; мы фактически пропускаем через программу «поток» строк. (И да, `std::getline` поддерживает диспетчеры памяти; если понадобится избежать использования динамической памяти, можно заменить `std::string` на `std::pmr::string`.) Функция `process` может применять регулярное выражение к получаемой строке (см. главу 10 «Регулярные выражения») для ее проверки и разделения на поля для дальнейшего парсинга.

Для чтения отдельных слов вместо строк можно взять за основу следующий фрагмент кода (если вы уверены, что определение `isspace` в текущей локали действительно служит разделителем слов):

```
template<class T>
struct streamer {
  std::istream& m_in;
  explicit streamer(std::istream& in) : m_in(in) {}
  auto begin() const
    { return std::istream_iterator<T>(m_in); }
  auto end() const
    { return std::istream_iterator<T>{}; }
};

int main()
{
  for (auto word : streamer<std::string>(std::cin)) {
    process(word);
  }
}
```

`std::istream_iterator<T>` – тип из стандартной библиотеки, объявленный в заголовке `<iterator>`, который обертывает указатель на `istream`. Функция `operator++` указателя читает значение типа `T` из `istream`, подобно `operator>>`, и затем это значение возвращается функцией `operator*` итератора. Объединив все это, можно прочитать всю последовательность слов, разделенных пробелами, из `std::cin`, положившись на тот факт, что `std::istream::operator>>(std::string&)` читает единственное слово, ограниченное пробелами.

Например, вот как можно воспользоваться нашим шаблонным классом `streamer`, чтобы прочитать последовательность целых чисел из `std::cin` и выполнить какие-то операции с каждым из них:

```
// Удвоить каждое число, введенное пользователем
for (auto value : streamer<int>(std::cin)) {
  printf("%d\n", 2*value);
}
```

Несмотря на высокую сложность средств ввода/вывода в C++, как и подобает языку системного программирования, уходящему корнями в 1980-е, из последних нескольких примеров видно, что эту сложность можно спрятать под слоем абстракции и в конечном итоге получить код, который выглядит так же просто, как код на Python.

Итоги

Вывод данных можно разделить на *форматирование* и *буферизацию*. Ввод данных можно разделить на *буферизацию* и *парсинг*; при этом этап парсинга можно упростить, если перед ним добавить этап *лексемизации*. (Более подробно о лексемизации мы поговорим в следующей главе!)

Классический iostreams API построен поверх `<stdio.h>`, который, в свою очередь, основан на API файловых дескрипторов POSIX. Нельзя понять верхние уровни, не понимая, как действуют нижние. В частности, разобраться в мешанине строк режимов `fopen` и флагов конструктора `fstream` можно, только обратившись к таблице, описывающей соответствия между ними и флагами POSIX-функции `open`.

POSIX API поддерживает только перемещение блоков данных в файловые дескрипторы и из файловых дескрипторов; он не предполагает возможность «буферизации» в обычном понимании этого слова. `<stdio.h>` API добавляет поверх POSIX API слой буферизации; объект `FILE` может работать в режиме полной буферизации, построчной буферизации или без буферизации. Кроме того, `<stdio.h>` предоставляет эффективные (но зависящие от локали) процедуры форматирования, наиболее важными из которых являются `fprintf`, `snprintf` и `sscanf`.

`<iostream>` API отделяет «streambuf» (определяет источник или приемник байтов и режим буферизации) от «потока данных» (который хранит настройки форматирования). Разные виды потоков данных (ввода/вывода, файл/строка) образуют классическую полиморфную иерархию со сложными и порой малопонятными отношениями наследования. В промышленном коде лучше избегать `<iostream>`, из-за меньшей производительности и прозрачности, в сравнении с интерфейсами `<stdio.h>` и POSIX. В любом случае, остерегайтесь процедур форматирования, зависящих от региональных настроек (локали).

Для однократных и коротких задач парсинга предпочтительнее использовать алгоритм `std::stoi`, который определяет широкий диапазон ошибок, а для форматирования – `std::to_string` или `snprintf`. Для сценариев, где требуется высокая производительность, парсинг лучше выполнять с помощью `std::from_chars`, а форматирование – с помощью `std::to_chars`, если они присутствуют в вашей стандартной библиотеке, в заголовке `<charconv>`.

Глава 10

Регулярные выражения

В предыдущей главе мы познакомились с форматированием ввода и вывода в C++. Мы увидели, что существуют хорошие решения форматирования вывода – при условии использования локали C, – и, несмотря на большое количество инструментов парсинга ввода, даже простая задача парсинга строки в целое число может быть сопряжена с большими сложностями. (Напомню, что из двух наиболее надежных методов `std::stoi(x)` требует преобразования `x` в значение типа `std::string`, который использует динамическую память, а `std::from_chars(x.begin(), x.end(), &value, 10)` пока присутствует не во всех реализациях стандартной библиотеки C++17.) Самое сложное в парсинге чисел – выяснить, что делать с остальной частью ввода, которая не является числом!

Парсинг можно упростить, если разбить его на две подзадачи: выяснить точно, какая часть ввода соответствует «одному элементу» (эту подзадачу обычно называют *лексемизацией*), и выполнить парсинг этого элемента, предусмотрев некоторую обработку ошибок, если значение элемента окажется за пределами некоторого диапазона или как-то иначе будут лишено смысла. В отношении ввода целых чисел лексемизация соответствует поиску наибольшей последовательности цифр во вводе и ее парсингу в числовое значение.

Регулярные выражения – это инструмент, поддерживаемый многими языками программирования, который решает задачу лексемизации не только для последовательностей цифр, но и для любых других входных форматов. Поддержка регулярных выражений стала частью стандартной библиотеки C++ в 2011 году и определяется в заголовке `<regex>`. В этой главе мы покажем вам, как использовать регулярные выражения для упрощения наиболее типичных задач парсинга.

Имейте в виду, что применение регулярных выражений может оказаться избыточным для *многих* задач парсинга. Во многих случаях они действуют слишком медленно и неизбежно требуют распределения памяти из кучи (то есть типы данных, представляющие регулярные выражения, не поддерживают возможность выбора диспетчеров памяти, о которых рассказывалось в главе 8 «Диспетчеры памяти»). Вместе с тем регулярные выражения незаменимы в задачах парсинга, где рукописный код синтаксического анализа в любом случае будет выполняться медленно; и в очень простых задачах, где удобочитаемость и надежность регулярных выражений перевешивают их недостаточно высо-

кую производительность. Проще говоря, поддержка регулярных выражений еще больше сблизила C++ с такими языками, как Python и Perl, в удобстве использования.

В этой главе вы:

- познакомитесь с «модифицированным ECMAScript», диалектом регулярных выражений в C++;
- узнаете, как сопоставлять, искать и заменять подстроки с помощью регулярных выражений;
- увидите дополнительные опасности недействительных итераторов;
- узнаете, каких особенностей регулярных выражений следует избегать.

Что такое регулярное выражение?

Регулярное выражение – это способ записи правил распознавания строк байтов или символов как принадлежащих (или не принадлежащих) определенному «языку». Под «языком» может подразумеваться все, что угодно, от «множества всех последовательностей цифр» до «множества всех ключевых слов C++». По сути, «язык» – это лишь правило деления мира строк на два множества – множество строк, соответствующих языку, и множество строк, *не* соответствующих ему.

Некоторые языки следуют относительно простым правилам, которые можно определить как *конечный автомат*, компьютерную программу, вообще не использующую память – программный счетчик и указатель, который сканирует ввод, выполняя только один проход. Язык «последовательность цифр» определенно относится к категории языков, которые можно выразить с помощью конечных автоматов. Такие языки мы называем *регулярными*.

Существуют также нерегулярные языки. Типичным примером нерегулярного языка является «допустимое арифметическое выражение», или, если свести его к сути: «правильное соотношение открывающих и закрывающих круглых скобок». Любая программа, способная отличить правильное соотношение открывающих и закрывающих круглых скобок `(((())))` от неправильного `(((()))` и `((((()))))`, в своей основе должна выполнять «подсчет», чтобы выявить непарность. Подсчет нельзя выполнить без применения изменяемой переменной или стека; поэтому «правильное соотношение открывающих и закрывающих круглых скобок» не является регулярным языком.

Как оказывается, для любого регулярного языка можно написать простое и ясное представление конечного автомата, распознающего его, которое, конечно же, одновременно является представлением правил самого языка. Мы называем такое представление *регулярным выражением*. Стандартная форма записи регулярных выражений была разработана еще в 1950-х и окончательно установилась в конце 1970-х, в таких программах для Unix, как `grep` и `sed`, которые следует изучать и в наши дни, но о которых не будет рассказываться в этой книге.

Стандартная библиотека C++ поддерживает несколько разных «диалектов» регулярных выражений, но по умолчанию (и поэтому вы должны использовать его всегда) используется диалект, заимствованный из стандарта ECMAScript – языка, больше известного как JavaScript, – с незначительными изменениями, касающимися конструкций в квадратных скобках. Описание синтаксиса регулярных выражений ECMAScript вы найдете в конце этой главы; но если вам приходилось пользоваться программой `grep`, вы без труда сможете следовать за примерами, не обращаясь к описанию.

Замечание об экранировании обратными слешами

В этой главе часто будут встречаться строки и регулярные выражения, содержащие символы обратного слеша (\). Как вы наверняка знаете, чтобы в C++ записать строковый литерал с обратным слешем, вы должны экранировать его еще одним обратным слешем: то есть строка `"\n"` представляет символ перевода строки, а `"\\n"` – строку с двумя символами, «обратный слеш», за которым следует символ «n». Обычно такие вещи легко отслеживаются, но в этой главе мы будем уделять обратным слешам особое внимание. Реализация регулярных выражений встроена непосредственно в библиотеку; то есть если вы запишете вызов `std::regex("\n")`, библиотека увидит «регулярное выражение», содержащее единственный пробельный символ, а если вы запишете вызов `std::regex("\\n")`, библиотека увидит строку из двух символов, начинающуюся с обратного слеша, которая *будет интерпретироваться библиотекой* как двухсимвольная экранированная последовательность, обозначающая «перевод строки». Если вам потребуется передать в библиотеку представление обозначения перевода, вы должны ввести трехсимвольную строку `\\\n`, которая в исходном коде на C++ записывается как пятисимвольная строка `"\\\n"`.

Возможно, в предыдущем абзаце вы заметили решение, которое я собираюсь использовать в этой главе. Когда я говорю о *строковом литерале* или значении в C++, я заключаю его в двойные кавычки, например: `"cat"`, `"a\\.b"`. Когда я говорю о *регулярном выражении*, которое вводится в текстовом редакторе или в клиенте электронной почты, или вручную передается библиотеке для выполнения, я записываю его без кавычек: `cat`, `a\.b`. Просто запомните: если вы видите строку без кавычек, представляющую литеральную последовательность символов, и хотите поместить ее в строковый литерал C++, вы должны удвоить обратные слеши, например превратив `a\.b` в `std::regex("a\\.b")`.

Я уже слышу ваш вопрос: «А как насчет низкоуровневых строковых литералов?» Низкоуровневые строковые литералы появились в C++11 и позволяют записать последовательность символов `a\.b`, «экранировав» всю строку целиком символом `R` и некоторыми скобками, например `R"(a\.b)"`, вместо экранирования каждого обратного слеша. Если ваша строка содержит круглые скобки, можно проявить изобретательность и записать любую произвольную строку перед открывающей и после закрывающей скобки, например: `R"fancy(a\.b)fancy"`. Низкоуровневые строковые литералы, такие как этот, могут включать

любые символы – обратные слеши, кавычки и даже переводы строк – при условии, что в них отсутствует закрывающая последовательность `)fancy"` (и если вы считаете, что такая последовательность может содержаться в строке, тогда просто выберите новую произвольную строку, такую как `)supercalifragilisticexpialidocious"`).

Синтаксис низкоуровневых строковых литералов в C++, начинающихся с символа `R`, напоминает синтаксис аналогичных литералов в Python (начинающихся с символа `r`). В Python строка `r"a\.b"` представляет строковый литерал `a\.b`; и является распространенным и идиоматическим средством представления регулярных выражений в исходном коде в виде строк, таких как `r"abc"`, даже не содержащих специальных символов. Но обратите внимание на важное отличие `r"a\.b"` от `R"(a\.b)"` – версия в C++ имеет дополнительны скобки! А скобки – важный специальный символ в грамматике регулярных выражений. В C++ строковые литералы `"(cat)"` и `R"(cat)"` отличаются как день и ночь – первый представляет пятисимвольное регулярное выражение `(cat)`, а второй – трехсимвольное `cat`. Если вы запишете `R"(cat)"`, имея в виду `"(cat)"` (эквивалентно `R"((cat))"`), в вашей программе появится очень тонкая ошибка. Еще более садистский пример: `R"a*(b*)a*"` – это допустимое регулярное выражение с удивительным смыслом! Поэтому я советую с большой осторожностью использовать низкоуровневые строковые литералы для записи регулярных выражений; обычно проще и безопаснее удвоить *все* обратные слеши, чем заботиться об удвоении *внешних* скобок.

Низкоуровневые строковые литералы удобно использовать для создания того, что в других языках называют «вложенными документами» (heredocs):

```
void print_help() {
  puts(R"(Специальные символы регулярных выражений:
  \ - экранирование
  | - разделение альтернатив
  . - соответствует любому символу
  [] - класс или множество символов
  () - сохраняющие круглые скобки или опережающая проверка
  ?*+ - "ноль или один ", "ноль или больше", "один или больше"
  {} - "точно N" или "от M до N" повторений
  ^$ - начало и конец "строки"
  \b - граница слова
  \d \s \w - цифра, пробел и слово (word)
  (?=foo) (?!foo) - опережающая проверка; негативная опережающая проверка
)");
```

То есть низкоуровневые литералы строк являются единственным типом строковых литералов в C++, в котором можно использовать переводы строк без всякого экранирования. Это удобно для вывода длинных сообщений или, например, для представления заголовков HTTP; но применение низкоуровневых литералов строк с их особым отношением к круглым скобкам делает их опасными для регулярных выражений. Поэтому я не буду использовать их в книге.

Воплощение регулярных выражений в объектах std::regex

Чтобы воспользоваться регулярным выражением в C++, недостаточно просто указать строку, такую как `"c[a-z]*t"`, – из этой строки нужно сконструировать *объект регулярного выражения* типа `std::regex`, а потом передать его *функции сопоставления*, такой как `std::regex_match`, `std::regex_search` или `std::regex_replace`. Каждый объект типа `std::regex` формирует конечный автомат для данного выражения, а для его конструирования требуется большой объем вычислений и памяти в куче. То есть если одно и то же регулярное выражение предполагается использовать много раз для сопоставления с входным текстом, библиотека дает нам удобную возможность выполнить дорогостоящие операции только один раз. С другой стороны, это означает, что создание и копирование объектов `std::regex` обходится довольно дорого, то есть конструирование объекта регулярного выражения внутри глубокого цикла – верный способ убить производительность программы:

```
std::regex rx("(left|right) ([0-9]+)");
// Объект регулярного выражения "rx" конструируется за пределами цикла.
std::string line;
while (std::getline(std::cin, line)) {
  // Внутри цикла многократно используется один и тот же объект "rx".
  if (std::regex_match(line, rx)) {
    process_command(line);
  } else {
    puts("Unrecognized command.");
  }
}
```

Имейте в виду, что объекты `regex` имеют семантику значений; «сопоставляя» входную строку с регулярным выражением, мы не изменяем сам объект `regex`. Регулярное выражение не запоминает найденных совпадений. Поэтому, чтобы извлечь результат сопоставления, например: «какая была дана команда, влево или вправо? на сколько шагов предписано переместиться?», – мы должны ввести новую сущность, которую можно изменять.

Объект `regex` предлагает следующие методы:

`std::regex(str, flags)` конструирует новый объект `std::regex`, транслируя (или «компилируя») заданную строку `str` в конечный автомат. Параметры, влияющие на процесс компиляции, передаются в битовой маске `flags`, в которой можно передать следующие флаги:

- `std::regex::icase`: интерпретировать все алфавитные символы без учета регистра;
- `std::regex::nosubs`: интерпретировать все группы в круглых скобках как несохраняющие;

- `std::regex::multiline`: якорный метасимвол ^ (и $) должен находить совпадение непосредственно после (и перед) символа `"\n"` во входной строке, а не только в начале (и в конце) входного текста.

Существуют также другие флаги, которые можно добавить по ИЛИ в аргумент `flags`. Но они или изменяют «диалект» регулярных выражений, делая его отличным от ECMAScript, за счет добавления недостаточно хорошо документированных и проверенных особенностей (`basic`, `extended`, `awk`, `grep`, `egrep`), или вносят зависимость от локали (`collate`), или просто ничего не делают (`optimize`). Поэтому в промышленном коде их лучше избегать.

Обратите внимание: даже притом, что процесс преобразования строки в объект `regex` часто называют «компиляцией регулярного выражения», он остается динамическим и протекает во время выполнения программы, когда вызывается конструктор `regex`, а не во время компиляции программы на C++. Если вы допустите синтаксическую ошибку в регулярном выражении, она обнаружится только во время выполнения – конструктор `regex` возбудит исключение типа `std::regex_error`, которое является подклассом `std::runtime_error`. Надежный код также должен быть готов получить от конструктора `regex` исключение `std::bad_alloc`; напомню, что `std::regex` не поддерживает возможности выбора диспетчера памяти.

`rx.mark_count()` возвращает количество сохраняющих групп в круглых скобках. Имя этого метода происходит от фразы «marked subexpression» (выделенное подвыражение), синонима «capturing group» (сохраняющая группа).

`rx.flags()` возвращает битовую маску, которая была передана конструктору.

Сопоставление и поиск

Узнать, соответствует ли заданная строка `haystack` заданному регулярному выражению `rneedle`, можно с помощью вызова `std::regex_match(haystack, rneedle)`. Объект регулярного выражения всегда передается как последний аргумент, по аналогии с синтаксисом вызова `haystack.match(rneedle)` в JavaScript и `haystack =~ rneedle` в Perl, но в противоположность синтаксису вызова `re.match(rneedle, haystack)` в Python. Функция `regex_match` возвращает `true`, если регулярному выражению соответствует вся строка, и `false` в противном случае:

```
std::regex rx("(left|right) ([0-9]+)");
std::string line;
while (std::getline(std::cin, line)) {
  if (std::regex_match(line, rx)) {
    process_command(line);
  } else {
    printf("Unrecognized command '%s'.\n", line.c_str());
  }
}
```

Функция `regex_search` возвращает `true`, если регулярному выражению соответствует любая часть входной строки. Фактически она как бы добавляет `.*` с обеих сторон регулярного выражения и затем выполняет алгоритм `regex_match`; но обычно реализации `regex_search` действуют намного быстрее, чем если бы в них выполнялась компиляция совершенно нового регулярного выражения.

Чтобы выполнить сопоставление только с частью символов в буфере (например, когда исходные данные извлекаются из сетевого соединения или из файла), в `regex_match` и `regex_search` можно передать пару итераторов, как описывалось в главе 3 «Алгоритмы с парами итераторов». Следующий пример просматривает байты только в пределах диапазона `[p, end)` и не требует, чтобы «строка» `p` завершалась нулевым символом:

```
void parse(const char *p, const char *end)
{
  static std::regex rx("(left|right) ([0-9]+)");
  if (std::regex_match(p, end, rx)) {
    process_command(p, end);
  } else {
    printf("Unrecognized command '%.*s'.\n",
    int(end - p), p);
  }
}
```

Этот интерфейс напоминает функцию `std::from_chars`, виденную нами в главе 9 «Потоки ввода/вывода».

Извлечение совпадений с подвыражениями

Чтобы использовать регулярные выражения для разделения входного потока на лексемы, нужна возможность извлечения подстрок, соответствующих всем сохраняющим группам. Такую возможность в C++ дает *объект совпадения* (match object) типа `std::smatch`. Да, это не опечатка! Объект совпадения действительно называется `smatch`. Это имя происходит от `std::string` match; существует также объект `cmatch`, от `const char *` match. Объекты `smatch` и `cmatch` отличаются типами поддерживаемых итераторов: `smatch` поддерживает `string::const_iterator`, а `cmatch` – `const char *`.

Создав пустой объект `std::smatch`, его нужно передать вторым параметром в вызов `regex_match` или `regex_search`. Эти функции «заполнят» объект `smatch` информацией о совпадениях, *если* таковые будут найдены. Если сопоставление потерпит неудачу, объект `smatch` опустошится или останется пустым.

Вот пример использования `std::smatch` для получения подстрок, описывающих направление и расстояние в нашей «команде для робота»:

```
std::pair<std::string, std::string>
parse_command(const std::string& line)
{
  static std::regex rx("(left|right) ([0-9]+)");
```

```
  std::smatch m;
  if (std::regex_match(line, m, rx)) {
    return { m[1], m[2] };
  } else {
    throw "Unrecognized command!";
  }
}

void test() {
  auto [dir, dist] = parse_command("right 4");
  assert(dir == "right" && dist == "4");
}
```

Обратите внимание: здесь, чтобы избежать создания («компиляции») нового объекта регулярного выражения в каждом вызове функции, использован статический объект `regex`. Для сравнения приводится тот же код, но уже использующий `const char *` и `std::cmatch`:

```
std::pair<std::string, std::string>
parse_command(const char *p, const char *end)
{
  static std::regex rx("(left|right) ([0-9]+)");
  std::cmatch m;
  if (std::regex_match(p, end, m, rx)) {
    return { m[1], m[2] };
  } else {
    throw "Unrecognized command!";
  }
}

void test() {
  char buf[] = "left 20";
  auto [dir, dist] = parse_command(buf, buf + 7);
  assert(dir == "left" && dist == "20");
}
```

В обоих случаях самое интересное происходит в строке с инструкцией `return`. Обнаружив совпадения с регулярным выражением во входной строке, можно потребовать от объекта совпадения `m` отыскать фрагменты, совпавшие с сохраняющими группами в регулярном выражении. Первая сохраняющая группа (в данном примере `(left|right)`) соответствует элементу `m[1]`, вторая группа (в данном примере `([0-9]+)`) соответствует элементу `m[2]` и т. д. Обращение к несуществующей группе, такой как `m[3]` в этом примере, вернет пустую строку; обращения к объекту совпадения никогда не возбуждают исключений.

Группа `m[0]` – особый случай: она ссылается на всю совпавшую последовательность. Если объект совпадения заполнен функцией `std::regex_match`, его элемент `m[0]` всегда будет содержать полную входную строку; если объект совпадения заполнен функцией `std::regex_search`, его элемент `m[0]` будет содержать только часть строки, совпавшую с регулярным выражением.

Существуют также две именованные группы: `m.prefix()` и `m.suffix()`. Они ссылаются на последовательности, *не* являющиеся частью совпадения – до и после совпавшей подстроки соответственно. Если совпадение в строке обнаружено, тогда сумма `m.prefix()` + `m[0]` + `m.suffix()` будет представлять полную входную строку.

Все эти «группы» представлены не объектами `std::string` – это было бы слишком дорого, – а легковесными объектами типа `std::sub_match<It>` (где `It` – либо `std::string::const_iterator`, либо `const char *`, как отмечалось выше). Все объекты `sub_match` неявно преобразуются в `std::string`, а в иных случаях ведут себя как `std::string_view`: их можно сравнивать со строковыми литералами, определять их длину и даже выводить в потоки C++ с помощью `operator<<`, не выполняя явного преобразования в `std::string`. Это решение имеет тот же недостаток, что и итераторы, ссылающиеся на контейнер, но не владеющие им: есть риск получить недействительные итераторы:

```
static std::regex rx("(left|right) ([0-9]+)");
std::string line = "left 20";
std::smatch m;
std::regex_match(line, m, rx);
  // m[1] теперь хранит итераторы на line

line = "hello world";
  // для line выделяется новый буфер в памяти

std::string oops = m[1];
  // результат этого вызова неопределен, потому что
  // итератор недействителен
```

Рассматривая предыдущий фрагмент кода, у вас может появиться опасение, что неявное преобразование (например, из `const char *` в `std::string`) может породить ошибку недействительности итератора в казалось бы безобидном коде. Взгляните на следующий фрагмент:

```
static std::regex rx("(left|right) ([0-9]+)");
std::smatch m;
std::regex_match("left 20", m, rx);
  // по логике m[1] должен сохранить итераторы на временную
  // строку, то есть они ИЗНАЧАЛЬНО недействительны.
  // К счастью, эта перегруженная версия удалена.
```

К счастью, разработчики стандартной библиотеки предвидели эту скрытую проблему и устранили ее, определив специальную перегруженную версию `regex_match(std::string&&, std::smatch&, const std::regex&)` как явно удаленную (с помощью того же синтаксиса `=delete`, которым вы пользуетесь для удаления нежелательных функций-членов). Это гарантирует, что предыдущий невинный код не скомпилируется и не станет источником ошибки получения недействительного итератора.

И все же итератор может стать недействительным, как в примере выше; чтобы не попасть в эту ловушку, вы должны рассматривать объекты `smatch` как временные сущности, подобно лямбда-выражению `[&]`, сохраняющему ссылку на мир. Заполнив объект `smatch`, вы не должны прикасаться ни к чему из его окружения, пока не извлечете необходимые элементы!

А теперь перечислим методы, предлагаемые объектами `smatch` и `cmatch`:

- `m.ready()`: `true`, если `m` был заполнен после его создания;
- `m.empty()`: `true`, если `m` представляет сопоставление, не увенчавшееся успехом (то есть если он был заполнен последним вызовом `regex_match` или `regex_search`, потерпевшим неудачу); `false`, если `m` представляет успешное сопоставление;
- `m.prefix()`, `m[0]`, `m.suffix()`: объекты `sub_match`, представляющие начало строки, не попавшее в совпадение, совпадение и конец строки, не попавший в совпадение (если `m` представляет сопоставление, не увенчавшееся успехом, ни один из этих объектов не имеет смысла);
- `m[k]`: объект `sub_match`, представляющий часть исходной строки, совпавшую с *k*-й сохраняющей группой; `m.str(k)` – удобное сокращение, эквивалентное `m[k].str()`;
- `m.size()`: ноль, если `m` представляет сопоставление, не увенчавшееся успехом; иначе число, на единицу больше числа сохраняющих групп в регулярном выражении, успешное совпадение с которым представляет `m`. Обратите внимание, что `m.size()` всегда согласуется с `operator[]`; диапазон значимых объектов `sub_match` всегда от `m[0]` до `m[m.size()-1]`;
- `m.begin()`, `m.end()`: итераторы для обхода содержимого объекта совпадения в `for`.

Объект `sub_match` предлагает следующие методы:

- `sm.first`: итератор, указывающий на начало совпадения во входной строке;
- `sm.second`: итератор, указывающий на конец совпадения во входной строке;
- `sm.matched`: `true`, если `sm` получен в результате успешного сопоставления; `false`, если `sm` является частью альтернативной ветви, которая не участвовала в сопоставлении; например, для регулярного выражения `(a)|(b)` и строки `"a"` мы получим: `m[1].matched && !m[2].matched`; а для строки `"b"` получим: `m[2].matched && !m[1].matched`;
- `sm.str()`: совпавшая подстрока, извлеченная из входной строки и преобразованная в `std::string`;
- `sm.length()`: длина совпавшей подстроки (`second - first`); эквивалент вызова `sm.str().length()`, но выполняется намного быстрее;
- `sm == "foo"`: сравнение с `std::string`, `const char *` или единственным символом; эквивалент вызова `sm.str() == "foo"`, но выполняется намного быстрее. К сожалению, в стандартной библиотеке C++17 нет перегруженной функции `operator==`, принимающей `std::string_view`.

Возможно, вам никогда не придется использовать это на практике, тем не менее есть возможность создать объект `smatch` или `sub_match`, хранящий итераторы на контейнеры, отличные от `std::string` или буферов символов. Например, ниже приводится все та же наша функция, выполняющая поиск совпадения с регулярным выражением в `std::list<char>` – глупо, но она работает!

```
template<class Iter>
std::pair<std::string, std::string>
parse_command(Iter begin, Iter end)
{
  static std::regex rx("(left|right) ([0-9]+)");
  std::match_results<Iter> m;
  if (std::regex_match(begin, end, m, rx)) {
    return { m.str(1), m.str(2) };
  } else {
    throw "Unrecognized command!";
  }
}

void test() {
  char buf[] = "left 20";
  std::list<char> lst(buf, buf + 7);
  auto [dir, dist] = parse_command(lst.begin(), lst.end());
  assert(dir == "left" && dist == "20");
}
```

Преобразование совпадений в значения данных

Просто, чтобы закрыть тему парсинга, ниже приводится пример извлечения строковых и целочисленных значений из совпадений, чтобы затем использовать их для перемещения нашего робота:

```
int main()
{
  std::regex rx("(left|right) ([0-9]+)");
  int pos = 0;
  std::string line;
  while (std::getline(std::cin, line)) {
    try {
      std::smatch m;
      if (!std::regex_match(line, m, rx)) {
        throw std::runtime_error("Failed to lex");
      }
      int how_far = std::stoi(m.str(2));
      int direction = (m[1] == "left") ? -1 : 1;
      pos += how_far * direction;
      printf("Robot is now at %d.\n", pos);
    } catch (const std::exception& e) {
      puts(e.what());
      printf("Robot is still at %d.\n", pos);
```

```
    }
  }
}
```

Любая ошибочная или недопустимая строка породит наше собственное исключение с сообщением `"Failed to lex"` или исключение `std::out_of_range`, которое возбудит `std::stoi()`. Если добавить проверку на целочисленное переполнение перед изменением `pos`, получится надежный парсер входных строк.

Чтобы обеспечить возможность извлечения отрицательных чисел и направлений без учета регистра символов, достаточно изменить код, как показано ниже:

```
int main()
{
  std::regex rx("((left)|right) (-?[0-9]+)", std::regex::icase);
  int pos = 0;
  std::string line;
  while (std::getline(std::cin, line)) {
    try {
      std::smatch m;
      if (!std::regex_match(line, m, rx)) {
        throw std::runtime_error("Failed to lex");
      }
      int how_far = std::stoi(m.str(3));
      int direction = m[2].matched ? -1 : 1;
      pos += how_far * direction;
      printf("Robot is now at %d.\n", pos);
    } catch (const std::exception& e) {
      puts(e.what());
      printf("Robot is still at %d.\n", pos);
    }
  }
}
```

Итерации по нескольким совпадениям

Рассмотрим регулярное выражение `(?!\d)\w+`, которому соответствует один любой идентификатор на языке C++. Мы уже знаем, как с помощью `std::regex_match` узнать, *является* ли входная строка идентификатором C++, и как с помощью `std::regex_search` найти *первый* идентификатор в заданной строке. Но как быть, если потребуется отыскать *все* идентификаторы, имеющиеся в строке?

Идея состоит в том, чтобы вызывать `std::regex_search` в цикле. Но здесь возникает проблема из-за якорных метасимволов в «опережающей негативной проверке», таких как `^` и `\b`. Чтобы правильно реализовать цикл с `std::regex_search`, мы должны сохранить состояния этих якорей. `std::regex_search` (и `std::regex_match`) поддерживает такую возможность посредством фла-

гов – флагов, определяющих *начальное состояние* конечного автомата для данной конкретной операции сопоставления. В данном случае нам нужен флаг `std::regex::match_prev_avail`, который сообщает библиотеке, что итератор `begin`, представляющий начало входной строки, в действительности не является ее «началом» (то есть может не совпадать с ^), и, чтобы узнать предыдущий символ при поиске совпадения с `\b`, безопасно можно проверить `begin[-1]`:

```
auto get_all_matches(
  const char *begin, const char *end,
  const std::regex& rx,
  bool be_correct)
{
  auto flags = be_correct ?
    std::regex_constants::match_prev_avail :
    std::regex_constants::match_default;
  std::vector<std::string> result;
  std::cmatch m;
  std::regex_search(begin, end, m, rx);
  while (!m.empty()) {
    result.push_back(m[0]);
    begin = m[0].second;
    std::regex_search(begin, end, m, rx, flags);
  }
  return result;
}

void test() {
  char buf[] = "baby";
  std::regex rx("\\bb.");
    // получить первые две буквы в каждом слове, начинающемся с "b"
  auto v = get_all_matches(buf, buf+4, rx, false);
  assert(v.size() == 2);
    // ой, слог "by" распознается как находящийся в начале слова!
  v = get_all_matches(buf, buf+4, rx, true);
  assert(v.size() == 1);
    // слог "by" правильно распознается как часть слова "baby"
}
```

В предыдущем примере, когда выполняется условие `!be_correct`, каждый вызов `regex_search` интерпретируется как независимый, поэтому поиск по регулярному выражению `\bb.` не отличает первую букву в слове `"by"` от третьей буквы в слове `"baby"`. Но когда потом функции `regex_search` передается флаг `match_prev_avail`, она отступает на шаг назад, – буквально – чтобы проверить, является ли символ перед «by» символом «слова». Поскольку предыдущий символ `"a"` является символом слова, второй вызов `regex_search` правильно отвергает совпадение со слогом `"by"`.

Использование `regex_search` в цикле, как в данном примере, – простая задача.., если регулярное выражение не совпадает с пустой строкой! Если сопоставление с регулярным выражением завершается успехом и `m[0].length() == 0`,

возникает бесконечный цикл. Поэтому внутренний цикл в `get_all_matches()` в действительности должен выглядеть так:

```
while (!m.empty()) {
  result.push_back(m[0]);
  begin = m[0].second;
  if (begin == end) break;
  if (m[0].length() == 0) ++begin;
  if (begin == end) break;
  std::regex_search(begin, end, m, rx, flags);
}
```

Стандартная библиотека включает удобный «вспомогательный» тип `std::regex_iterator`, инкапсулирующий логику предыдущих фрагментов кода; использование `regex_iterator` может предохранить вас от появления некоторых тонких ошибок, связанных с совпадениями, имеющими нулевую длину. К сожалению, он не избавляет от необходимости вводить код и немного увеличивает вероятность столкнуться с недействительным итератором. `regex_iterator` параметризуется типом базового итератора, так же как `match_results`, поэтому, если сопоставление производится со строкой `std::string`, вы должны использовать `std::sregex_iterator`, если сопоставление производится с `const char *`, вы должны использовать `std::cregex_iterator`. Вот как можно переписать предыдущий пример в терминах `sregex_iterator`:

```
auto get_all_matches(
  const char *begin, const char *end,
  const std::regex& rx)
{
  std::vector<std::string> result;
  using It = std::cregex_iterator;
  for (It it(begin, end, rx); it != It{}; ++it) {
    auto m = *it;
    result.push_back(m[0]);
  }
  return result;
}
```

Подумайте, как в этом неуклюжем цикле можно было бы извлечь преимущества вспомогательного класса `streamer<T>` из примера в конце главы 9 «Потоки ввода/вывода».

Точно так же можно организовать обход совпадений с сохраняющими группами в каждом совпадении с регулярным выражением, вручную или с использованием «вспомогательного» библиотечного типа. Обход вручную мог бы выглядеть примерно так:

```
auto get_tokens(const char *begin, const char *end,
  const std::regex& rx)
{
  std::vector<std::string> result;
```

```
  using It = std::cregex_iterator;
  std::optional<std::csub_match> opt_suffix;
  for (It it(begin, end, rx); it != It{}; ++it) {
    auto m = *it;
    std::csub_match nonmatching_part = m.prefix();
    result.push_back(nonmatching_part);
    std::csub_match matching_part = m[0];
    result.push_back(matching_part);
    opt_suffix = m.suffix();
  }
  if (opt_suffix.has_value()) {
    result.push_back(opt_suffix.value());
  }
  return result;
}
```

Напомню, что `regex_iterator` – это всего лишь обертка вокруг `regex_search`, то есть в этом случае `m.prefix()` гарантированно хранит часть входной строки, предшествующую текущему совпадению, вплоть до конца предыдущего совпадения. Последовательно выделяя префиксы, предшествующие совпадению, сами совпадения и заканчивая специальным случаем несовпавшего суффикса, входную строку можно разбить на вектор «слов», чередующихся с «разделителями слов». Такой код легко видоизменить и сохранять только «слова» или только «разделители», или даже сохранять `m[1]` вместо `m[0]` и т. д.

Всю эту логику целиком инкапсулирует библиотечный тип `std::sregex_token_iterator`, но интерфейс его конструктора может показаться запутанным, если не знать, как эта задача решается вручную. Конструктор `sregex_token_iterator` принимает пару итераторов, регулярное выражение и *вектор индексов сохраняющих групп*, где индекс `-1` соответствует специальному случаю «префиксы (а также суффикс)».

```
auto get_tokens(const char *begin, const char *end,
  const std::regex& rx)
{
  std::vector<std::string> result;
  using TokIt = std::cregex_token_iterator;
  for (TokIt it(begin, end, rx, {-1, 0}); it != TokIt{}; ++it) {
    std::csub_match some_part = *it;
    result.push_back(some_part);
  }
  return result;
}
```

Если массив `{-1, 0}` заменить массивом `{0}`, тогда в вектор `result` попадут только фрагменты из входной строки, совпадающие с `rx`. Если заменить его массивом `{1, 2, 3}`, наш цикл увидит только совпадения с сохраняющими группами (`m[1]`, `m[2]` и `m[3]`) в каждом совпадении `m` с `rx`. Напомню, что из-за оператора альтернативы `|` сопоставление с группами может не выполняться,

и m[k].matched получит значение false. regex_token_iterator не пропускает эти результаты. Например:

```
std::string input = "abc123...456...";
std::vector<std::ssub_match> v;
std::regex rx("([0-9]+)|([a-z]+)");
using TokIt = std::sregex_token_iterator;
std::copy(
  TokIt(input.begin(), input.end(), rx, {1, 2}),
  TokIt(),
  std::back_inserter(v)
);
assert(!v[0].matched); assert(v[1] == "abc");
assert(v[2] == "123"); assert(!v[3].matched);
assert(v[4] == "456"); assert(!v[5].matched);
```

Одним из наиболее привлекательных случаев использования regex_token_iterator могло бы стать разбиение строки на «слова» по пробельным символам. К сожалению, это ненамного проще привычных подходов, таких как istream_iterator<string> (см. главу 9 «Потоки ввода/вывода») или strtok_r.

Использование регулярных выражений для замены строк

Программисты с опытом программирования на Perl или пользователи утилиты командной строки sed наверняка представляют регулярные выражения в первую очередь как средство для изменения строк – например, для «удаления всех подстрок, совпавших с регулярным выражением» или для «замены всех вхождений данного слова другим словом». Стандартная библиотека C++ поддерживает подобную замену в виде функции std::regex_replace. Она основана на JavaScript-методе String.prototype.replace и поддерживает собственный мини-язык форматирования.

std::regex_replace(str, rx, "replacement") возвращает строку std::string, являющуюся результатом поиска всех совпадений с rx в строке str и их замены подстрокой, которая в данном примере представлена литералом "replacement". Например:

```
std::string s = "apples and bananas";
std::string t = std::regex_replace(s, std::regex("a"), "e");
assert(t == "epples end benenes");
std::string u = std::regex_replace(s, std::regex("[ae]"), "u");
assert(u == "upplus und bununus");
```

Кроме того, комбинации с символом '$' в строке замены ("replacement") имеют особый смысл!

- `"$&"` замещается всей совпавшей подстрокой, `m[0]`. Обе библиотеки, libstdc++ и libc++, поддерживают комбинацию `"$0"` как нестандартный синоним для `"$&"`;
- `"$1"` замещается первым совпадением, `m[1]`; `"$2"` замещается вторым совпадением `m[2]` и т. д., вплоть до `"$99"`. Возможность сослаться на 100-е совпадение отсутствует. Комбинация `"$100"` соответствует «совпадению `m[10]`, за которым следует литерал `'0'`». Чтобы выразить «`m[1]`, за которым следует литерал `'0'`», нужно записать `"$010"`;
- `"$`"` (обратный апостроф) замещается результатом `m.prefix()`;
- `"$'"` (одиночная кавычка) замещается результатом `m.suffix()`;
- `"$$"` замещается символом доллара.

Обратите внимание на несимметричность комбинаций `"$`"` и `"$'"`. Это объясняется тем, что `m.prefix()` ссылается на часть строки между концом предыдущего совпадения и началом текущего, а `m.suffix()` всегда ссылается на часть строки между концом текущего совпадения и концом строки! Вам едва ли придется использовать комбинации `"$`"` и `"$'"` в своей практике.

Вот пример использования `regex_replace` для удаления из кода всех вхождений `std::` или их замены на `my::`:

```
auto s = "std::sort(std::begin(v), std::end(v))";
auto t = std::regex_replace(s, std::regex("\\bstd::(\\w+)"), "$1");
assert(t == "sort(begin(v), end(v))");
auto u = std::regex_replace(s, std::regex("\\bstd::(\\w+)"), "my::$1");
assert(u == "my::sort(my::begin(v), my::end(v))");
```

JavaScript-функция `String.prototype.replace` позволяет передавать произвольные функции вместо комбинаций со знаком доллара. Функция `regex_replace` в C++ пока не поддерживает такой возможности, но вы легко сможете написать свою версию:

```
template<class F>
std::string regex_replace(std::string_view haystack,
  const std::regex& rx, const F& f)
{
  std::string result;
  const char *begin = haystack.data();
  const char *end = begin + haystack.size();
  std::cmatch m, lastm;
  if (!std::regex_search(begin, end, m, rx)) {
    return std::string(haystack);
  }
  do {
    lastm = m;
    result.append(m.prefix());
    result.append(f(m));
    begin = m[0].second;
    begin += (begin != end && m[0].length() == 0);
```

```
    if (begin == end) break;
  } while (std::regex_search(begin, end, m, rx,
    std::regex_constants::match_prev_avail));
  result.append(lastm.suffix());
  return result;
}

void test()
{
  auto s = "std::sort(std::begin(v), std::end(v))";
  auto t = regex_replace(s, std::regex("\\bstd::(\\w+)"),
    [](auto&& m) {
      std::string result = m[1].str();
      std::transform(m[1].first, m[1].second,
      begin(result), ::toupper);
      return result;
    });
  assert(t == "SORT(BEGIN(v), END(v))");
}
```

С помощью этой усовершенствованной версии `regex_replace` можно выполнять весьма сложные операции, такие как «преобразование всех идентификаторов из змеиной_нотации в ВерблюжьюНотацию».

На этом мы завершаем краткий тур по инструментам в заголовке `<regex>`. В оставшейся части главы мы будем знакомиться с диалектом регулярных выражений ECMAScript. Я надеюсь, эта часть будет полезна читателям, которым прежде не приходилось использовать регулярные выражения, и поможет освежить память остальным.

Грамматика регулярных выражений ECMAScript

Диалект ECMAScript имеет простые правила чтения и записи регулярных выражений. Регулярное выражение – это всего лишь строка символов (такая как `a[bc].d*e`), которую следует читать слева направо. Большинство символов в таких строках представляют самих себя, то есть `cat` – это допустимое регулярное выражение, которому соответствует строка `"cat"`. Специальные символы, которые не представляют самих себя и позволяют создавать регулярные выражения, более интересные, чем `"cat"`, – это знаки препинания:

```
^ $ \ . * + ? ( ) [ ] { } |
```

`\` – если вы собираетесь использовать регулярные выражения для описания множества строк, включающих знаки препинания, вы можете использовать обратный слеш для экранирования этих специальных символов. Например, `\$42\.00` – это регулярное выражение, совпадающее с единственной строкой `"$42.00"`. Но самое интересное обратный слеш можно использовать для превращения обычных символов в специальные! `n` – это регулярное выражение,

совпадающее с буквой "n", а \n – регулярное выражение, совпадающее с символом перевода строки. d – регулярное выражение, совпадающее с буквой "d", а \d – регулярное выражение, совпадающее с любой цифрой, то есть эквивалент [0-9].

Вот полный список комбинаций с символом обратного слеша, поддерживаемых грамматикой регулярных выражений в C++:

- \1, \2, ... \10, ... – обратные ссылки (желательно избегать);
- \b – граница слова, а \B можно использовать взамен (?!\b);
- \d – аналог [[:digit:]], а \D – аналог [^[:digit:]];
- \s – замена для [[:space:]], а \S – для [^[:space:]];
- \w – аналог [0-9A-Za-z_], а \W – аналог [^0-9A-Za-z_];
- \c*X* – разные «управляющие символы» (желательно избегать);
- \x*XX* – шестнадцатеричный код символа в привычном понимании;
- \u00*XX* – код символа Юникода в привычном понимании;
- \0, \f, \n, \r, \t, \v – имеют привычное значение.

Точка (.) – специальный символ, имеющий смысл «любой один символ». Например, a.c – это допустимое регулярное выражение, совпадающее с такими строками, как "aac", "a!c" и "a\0c". При этом точка (.) никогда не совпадает с символом перевода строки или возврата каретки; и, так как регулярные выражения в C++ работают на уровне байтов, а не символов Юникода, точка (.) будет совпадать только с одним байтом (кроме '\\n' и '\\r') и никогда с последовательностью из нескольких байтов, даже если она представляет допустимый кодовый пункт UTF-8.

[] – группа символов в квадратных скобках представляет «точно один символ из данного набора», то есть c[aou]t – это допустимое регулярное выражение, совпадающее со строками "cat", "cot" и "cut". Квадратные скобки можно использовать для «экранирования» большинства символов; например, [$][.][*][+][?][(][)][[][{][}][|] – регулярное выражение, совпадающее с единственной строкой "$.*+?(){}|". Однако квадратные скобки нельзя использовать для экранирования символов], \ и ^.

[^] – группа символов в квадратных скобках, начинающаяся с символа ^, представляет «точно один символ не из данного набора», то есть c[^aou]t совпадет с "cbt" или "c^t", но не совпадет с "cat". Диалект ECMAScript поддерживает также тривиальные случаи [] и [^]; [] означает «точно один символ из данного пустого набора» (то есть не совпадет ни с чем), а [^] означает «точно один символ не из данного пустого набора» (то есть совпадет с любым одним символом – подобно точке (.), но лучше, потому что совпадет также с символом перевода строки или возврата каретки).

Кроме того, квадратные скобки [] придают специальное значение еще нескольким символам: дефис (-) внутри квадратных скобок, если он не первый и не последний, обозначает «диапазон» между соседними символами слева и справа от него. То есть ro[s-v]e – это регулярное выражение, совпадающее с четырьмя строками "rose", "rote", "roue" и "rove". Поддерживается воз-

можность записи нескольких распространенных диапазонов – которые можно найти в заголовке <ctype.h> – с использованием синтаксиса [:foo:]: [[:digit:]] – аналог [0-9], [[:upper:][:lower:]] – аналог [[:alpha:]] и [A-Za-z], и т. д.

Поддерживается также встроенный синтаксис вида [[.x.]] и [[=x=]]; он служит целям сравнения с учетом региональных настроек, и использовать его в промышленном коде крайне нежелательно. Просто имейте в виду, что если когда-нибудь вам понадобится включить символ [в класс в квадратных скобках, лучше всего экранировать его обратным слешем: оба регулярных выражения, foo[=([;] и foo[(\[=;], соответствуют строкам "foo=", "foo(", "foo[" и "foo;", но foo[([=;] – недопустимое регулярное выражение, и попытка скомпилировать его при создании объекта std::regex приведет к исключению во время выполнения.

+ – выражение или одиночный символ, непосредственно предшествующий знаку плюс (+), должно находить соответствие во входной строке любое положительное число раз. Например, регулярному выражению ba+ соответствуют строки "ba", "baa", "baaa" и т. д.

* – выражение или одиночный символ, непосредственно предшествующий звездочке (*), должно находить соответствие во входной строке любое число – даже ни разу! Например, регулярному выражению ba* соответствуют строки "ba", "baa" и "baaa", а также "b".

? – выражение или одиночный символ, непосредственно предшествующий знаку вопроса (?), должно находить соответствие во входной строке ноль или один раз. Например, регулярному выражению coo?t соответствуют только строки "cot" и "coot".

{n} – выражение или одиночный символ, непосредственно предшествующий фигурным скобкам с одним целым числом внутри, должно находить соответствие во входной строке точно указанное число раз. Например, регулярному выражению b(an){2}a соответствует строка "banana"; выражению b(an){3}a – строка "bananana".

{m,n} – выражение или одиночный символ, непосредственно предшествующий фигурным скобкам с двумя целыми числами m и n внутри, разделенными запятой, должно находить соответствие во входной строке от m до n раз (включительно). Например, регулярному выражению b(an){2,3}a соответствуют только строки "banana" и "bananana".

{m,} – отсутствие второго числа фактически означает бесконечность; то есть регулярное выражение x{42,} означает «42 и более совпадений с x» и эквивалентно выражению x{42}x*. Диалект ECMAScript не позволяет опускать число m.

| – два регулярных выражения можно «объединить» с помощью вертикальной черты (|), чтобы выразить идею «или-или». Например, регулярному выражению cat|dog соответствуют только строки "cat" и "dog"; а выражению (tor|shark)nado соответствуют строки "tornado" и "sharknado". Оператор | регулярных выражений имеет очень низкий приоритет, так же как и в выражениях на языке C++.

`()` – круглые скобки действуют так же, как в математике, формируя подвыражения, которые должны рассматриваться как одно целое. Например, `ba*` означает «символ `b` и ноль или более символов `a`», но `(ba)*` означает «ноль или более пар символов `ba`». То есть первому выражению соответствуют строки `"b"`, `"ba"`, `"baa"` и т. д., а версии с круглыми скобками – строки `""`, `"ba"`, `"baba"` и т. д.

Круглые скобки имеют еще и второе назначение – они применяются не только для *группировки*, но также для *сохранения* (или захвата) частей совпадения для последующей обработки. Для каждой открывающей скобки `(` в регулярном выражении создается свой объект `sub_match` в объекте `std::smatch` с результатом.

Если потребуется создать подвыражение без создания объекта `sub_match`, используйте синтаксис *несохраняющей* группировки `(?:foo)`:

```
std::string s = "abcde";
std::smatch m;
std::regex_match(s, m, std::regex("(a|b)*(.*)e"));
assert(m.size() == 3 && m[2] == "cd");
std::regex_match(s, m, std::regex("(?:a|b)*(.*)e"));
assert(m.size() == 2 && m[1] == "cd");
```

Несохраняющая группировка может пригодиться в некоторых контекстах; но потенциальному читателю вашего кода будет более понятно, если вы используете обычные сохраняющие скобки `()` и будете игнорировать ненужные объекты `sub_match`, чем если вы разбросаете по всему коду `(?:)` в попытке устранить неиспользуемые объекты `sub_match`. Избыточные (неиспользуемые) объекты `sub_match` очень дешевы с точки зрения производительности.

Непоглощающие конструкции

`(?=foo)` находит соответствие шаблону `foo` во входной строке и затем «откатывается» назад, в результате из входной строки не поглощается ни один символ (то есть, текущая позиция в строке остается на месте). Это называется «опережающей проверкой» (lookahead). Например, выражение `c(?=a)(?=a)(?=a)at` соответствует строке `"cat"`, а `(?=.*[A-Za-z])(?=.*[0-9]).*` соответствует любой строке, содержащей хотя бы один алфавитный символ и одну цифру.

`(?!foo)` – это «негативная опережающая проверка» (negative lookahead). Она заглядывает вперед в поисках совпадения с `foo` во входной строке и *считается совпавшей*, если совпадение с `foo` не найдено, и *несовпавшей* – если найдено. То есть, например, `(?!\d)\w+` соответствует любому идентификатору или ключевому слову языка C++ – любой последовательности алфавитно-цифровых символов, не начинающейся с цифры. Обратите внимание, что первый символ не должен соответствовать `\d` и он не поглощается конструкцией `(?!\d)`; он будет поглощен метасимволом `\w`. Внешне похожее регулярное выражение `[^0-9]\w+` может ошибочно принимать такие строки, как `"#xyzzy"`, за допустимые идентификаторы, хотя они таковыми не являются.

Обе конструкции, `(?=)` и `(?!)`, не только ничего не поглощают, но и не сохраняют, в точности как `(?:)`. Однако ничто не мешает использовать конструкцию

`(?=(foo))`, чтобы сохранить все или часть совпадения с опережающей проверкой.

`^` и `$` – символ крышки (`^`) вне квадратных скобок соответствует началу строки, сопоставляемой с регулярным выражением; а `$` соответствует ее концу. Их удобно использовать для «привязки» регулярного выражения к началу и/или концу строки в контексте `std::regex_search`. В регулярных выражениях `std::regex::multiline` крышка (`^`) и доллар (`$`) действуют как «ретроспективная» (lookbehind) и «опережающая» (lookahead) проверки соответственно:

```
std::string s = "ab\ncd";
std::regex rx("^ab$[^]^cd$", std::regex::multiline);

assert(std::regex_match(s, rx));
```

Объединив все вместе, можно написать регулярное выражение `foo[a-z_]+ (\d|$)` для поиска «букв `foo`, за которыми следует одна или более других букв и/или символов подчеркивания; за которыми следует цифра или конец строки».

Если вам требуется более глубокое освещение синтаксиса регулярных выражений, загляните на cppreference.com. А если этого окажется недостаточно – благодаря тому, что C++ фактически копирует диалект ECMAScript регулярных выражений, – воспользуйтесь любым руководством по регулярным выражениям в JavaScript! Вы можете даже проверять свои регулярные выражения в консоли своего браузера. Единственное отличие регулярных выражений в C++ и JavaScript – в C++ поддерживается синтаксис определения классов символов с двойными квадратными скобками, например `[[:digit:]]`, `[[.x.]]` и `[[=x=]]`, а в JavaScript – нет. В JavaScript такие регулярные выражения интерпретируются как `[\[:digt]\]`, `[\[.x]\]` и `[\[=x]\]` соответственно.

Малопонятные особенности и ловушки ECMAScript

Выше в этой главе я упоминал о некоторых особенностях `std::regex`, таких как `std::regex::collate`, `std::regex::optimize` и флаги, изменяющие диалект ECMAScript на какой-то другой, которыми лучше не пользоваться в промышленном коде. Но грамматика диалекта регулярных выражений ECMAScript тоже содержит несколько малопонятных особенностей, которых желательно избегать.

Обратный слеш, предшествующий одной или нескольким цифрам (кроме `\0`), создает *обратную ссылку* (backreference). Обратная ссылка `\1` соответствует «последовательности символов, совпавших с первой сохраняющей группой»; например, регулярному выражению `(cat|dog)\1` будут соответствовать строки `"catcat"` и `"dogdog"`, но не `"catdog"`, а выражению `(a*)(b*)c\2\1` будет соответствовать строка `"aabbbcbbbaa"`, но не `"aabbbcbbba"`. Обратные ссылки могут иметь очень странную семантику, особенно в комбинации с непоглоща-

ющими конструкциями, такими как `(?=foo)`, поэтому я советую избегать их по мере возможности.

Если у вас возникла проблема с обратными ссылками, прежде всего проверьте правильность экранирования обратных слешей. Не забывайте, что `std::regex("\1")` – это регулярное выражение, которому соответствует управляющий символ ASCII с кодом 1. Обратная ссылка в исходном коде должна иметь вид: `std::regex("\\1")`.

Использование обратных ссылок уводит вас из мира *регулярных языков* в более разнообразный мир *контекстно-зависимых языков*, а это значит, что библиотека будет вынуждена поступиться эффективностью алгоритма сопоставления на основе конечного автомата в пользу более мощного, но менее производительного алгоритма «с возвратами». Это еще одна причина сторониться обратных ссылок, если в них нет абсолютной необходимости.

Однако по состоянию на 2017 год многие производители реализаций стандартной библиотеки в действительности не переключают алгоритмы, независимо от отсутствия или наличия обратных ссылок в регулярном выражении; они всегда используют медленный алгоритм с возвратами. Кроме того, поскольку не нашлось ни одного производителя, пожелавшего реализовать отдельный алгоритм для диалектов `std::regex::awk` и `std::regex::extended`, не имеющих обратных ссылок, они используют для них все тот же алгоритм с возвратами! Аналогично большинство производителей реализует `regex_match(s, rx)` в терминах `regex_match(s, m, rx)` и затем просто отбрасывает избыточный `m`, вместо того чтобы создать потенциально более быстрый алгоритм для `regex_match(s, rx)`. Впрочем, подобные оптимизации могут появиться в библиотеке в ближайшие 10 лет, но я бы не задерживал дыхание в ожидании их появления.

Другая малопонятная особенность: *жадное* поведение по умолчанию квантификаторов `*`, `+` и `?`, например выражение `(a*)` постарается поглотить из строки как можно больше символов `a`. Жадный квантификатор можно превратить в *нежадный*[1], добавив после него знак вопроса (`?`); то есть выражение `(a*?)` поглотит из строки *минимальное* количество символов `a`. Это имеет значение, только если вы используете сохраняющие группы. Например:

```
std::string s = "abcde";
std::smatch m;
std::regex_match(s, m, std::regex(".*([bcd].*)e"));
assert(m[1] == "d");
std::regex_match(s, m, std::regex(".*?([bcd].*)e"));
assert(m[1] == "bcd");
```

[1] Жадные и нежадные квантификаторы иногда называют максимальными и минимальными. – *Прим. перев.*

В первом случае подвыражение `.*` жадно поглотит символы `abc`, оставив для совпадения с сохраняющей группой только символ `d`. Во втором случае подвыражение `.*?` нежадно поглотит только символ `a`, оставив `bcd` для сохраняющей группы. (В действительности подвыражение `.*?` могло бы удовлетвориться пустой строкой, не поглотив ни одного символа, но не может сделать этого, потому что иначе строка будет признана не соответствующей всему регулярному выражению.)

Обратите внимание, что синтаксис нежадности не следует «обычным» правилам композиции операторов. Согласно правилам в языке C++, можно было бы ожидать, что выражение `a+*` равноценно выражению `(a+)*` (впрочем, так оно и есть), а выражение `a+?` равноценно выражению `(a+)?` (в действительности это не так). Поэтому, увидев в регулярном выражении последовательность знаков препинания, задержитесь и рассмотрите ее внимательно – она может означать совсем не то, что могло бы показаться с первого взгляда!

Итоги

Регулярные выражения – отличное средство разделения входной строки на *лексемы* перед парсингом. По умолчанию в C++ используется тот же диалект регулярных выражений, что и в JavaScript. Не забывайте об этом преимуществе. Старайтесь избегать низкоуровневых строковых литералов там, где дополнительная пара круглых скобок может ввести в заблуждение. Всегда, когда это возможно, экранируйте специальные символы квадратными скобками вместо обратных слешей.

`std::regex rx` является практически неизменяемым объектом и представляет конечный автомат. `std::smatch m` – изменяемый объект и хранит информацию о конкретном совпадении в исходной строке. `sub_match m[0]` представляет подстроку, соответствующую всему регулярному выражению; `m[k]` представляет совпадение с *k*-й сохраняющей группой.

`std::regex_match(s, m, rx)` проверяет соответствие регулярному выражению всей исходной строки; `std::regex_search(s, m, rx)` отыскивает совпадение в исходной строке. Не забывайте, что строка передается первой, а регулярное выражение последним, в точности как в JavaScript и Perl.

`std::regex_iterator`, `std::regex_token_iterator` и `std::regex_replace` – несколько неудобные «вспомогательные» функции, основанные на `regex_search`. Прежде чем задумываться об их использовании, овладейте сначала всеми возможностями `regex_search`.

Остерегайтесь появления недействительных итераторов! Никогда не изменяйте и не уничтожайте объект регулярного выражения, если все еще имеется `regex_iterator`, ссылающийся на него; и никогда не изменяйте и не уничтожайте строку, на которую ссылается `smatch`.

Глава 11

Случайные числа

В предыдущей главе вы познакомились с регулярными выражениями, вошедшими в стандартную библиотеку C++ начиная с C++11, но все еще остающимися малоизвестными многим программистам. Вы увидели, что регулярные выражения могут пригодиться в двух ситуациях: в сложных программах, требующих надежного парсинга разнообразных входных форматов, и в тривиальных сценариях, когда важны удобочитаемость и скорость разработки.

Еще одна особенность библиотеки, полезная в этих же ситуациях, – механизмы *генерации случайных чисел*. Многие простые программы требуют применения случайных чисел то тут, то там, но программистов на C++ десятилетиями убеждали, что классическая функция `rand()` из libc – единственно верный выбор. С другой стороны, `rand()` абсолютно не подходит для нужд криптографии и сложного численного моделирования. Однако библиотеке `<random>` удалось удовлетворить все три потребности.

В этой главе рассматриваются следующие темы:

- отличия между истинно случайными числами и псевдослучайными числовыми последовательностями;
- отличия между *генератором* случайных битов и механизмом генерации значений с определенным *распределением*;
- три стратегии инициализации генератора случайных чисел;
- несколько генераторов и распределений в стандартной библиотеке и примеры их использования;
- как перетасовать колоду карт в C++17.

Случайные и псевдослучайные числа

Говоря о случайных числах в контексте компьютерного программирования, важно отличать по-настоящему случайные числа, получаемые из физически недетерминированного источника, и *псевдослучайные числа*, получаемые из алгоритмов, которые детерминировано производят поток чисел, «похожих на случайные». Такие алгоритмы называют **генераторами псевдослучайных чисел** (Pseudo-Random Number Generator, PRNG). Все генераторы псев-

дослучайных чисел действуют одинаково – они имеют некоторое внутреннее *состояние* и выполняют некоторые действия, когда пользователь запрашивает *следующее число*. Каждый раз, когда мы обращаемся за следующим числом, PRNG кодирует свое внутреннее состояние с применением некоторого детерминированного алгоритма и возвращает некоторую его часть. Например:

```
template<class T>
class SimplePRNG {
  uint32_t state = 1;
public:
  static constexpr T min() { return 0; }
  static constexpr T max() { return 0x7FFF; }

  T operator()() {
    state = state * 1103515245 + 12345;
    return (state >> 16) & 0x7FFF;
  }
};
```

Этот класс `SimplePRNG` реализует линейный конгруэнтный генератор, очень похожий на реализацию `rand()` в стандартной библиотеке. Обратите внимание, что `SimplePRNG::operator()` генерирует целые числа в 15-битном диапазоне [0, 32767], но внутреннее состояние хранит в 32-битном диапазоне. Эта закономерность соблюдается также в действующих PRNG. Например, стандартный алгоритм Mersenne Twister хранит почти 20 Кбайт внутреннего состояния! Наличие внутреннего состояния такого большого объема означает, что в кодировании можно использовать огромное количество битов и только малая часть внутреннего состояния PRNG будет «утекать» с каждым сгенерированным числом. Это затрудняет предсказание следующего вывода PRNG человеком (или компьютером) на основе небольшого количества предыдущих результатов. Трудность предсказания результатов позволяет назвать такой алгоритм *генератором псевдослучайных чисел*. Если бы вывод следовал некоторому шаблону и был легко предсказуем, мы назвали бы такой алгоритм генератором *неслучайных* чисел!

Несмотря на псевдослучайные качества, поведение PRNG всегда идеально детерминировано; он точно следует реализованному алгоритму. Если включить программу, использующую PRNG, и выполнить несколько попыток, в каждой запуская программу несколько раз подряд, мы ожидаемо получим в каждой попытке одну и ту же последовательность псевдослучайных чисел. Этот строгий детерминизм вынуждает нас назвать этот алгоритм генератором *псевдо*случайных чисел.

Другая особенность генераторов *псевдослучайных* чисел – два генератора, действующих по одному и тому же алгоритму, но немного отличающихся начальными состояниями, будут быстро *расходиться* и производить совершенно разные последовательности – так же, как две капли воды, попавшие на тыльную сторону вашей ладони, побегут в разные стороны. То есть чтобы программа

при каждом запуске производила разные последовательности псевдослучайных чисел, достаточно просто инициализировать PRNG разными *начальными состояниями*. Установку начального состояния PRNG называют *осеменением* (seeding) PRNG.

Существует, по меньшей мере, три стратегии инициализации PRNG.

- Использование начального состояния, полученного извне – от вызывающей программы или от конечного пользователя. Эта стратегия лучше всего подходит для случаев, когда требуется воспроизводимость результатов, например при моделировании методом Монте-Карло или в процессе модульного тестирования.
- Использование предсказуемого, но изменяющегося начального состояния, как, например, текущее время. До C++11 это была самая часто используемая стратегия, потому что стандартная библиотека C поддерживает переносимую и удобную функцию `time` и не предоставляет никаких других переносимых способов получения по-настоящему случайных битов. Инициализация на основе чего-то предсказуемого, как время, не подходит для задач, связанных с безопасностью. Начиная с C++11 вы не должны больше использовать эту стратегию.
- Использование *по-настоящему случайного* начального состояния, получаемого из некоторых платформенно-зависимых источников «истинно случайных» битов.

Истинно случайные биты собираются операционной системой по всем видам случайных событий; классический пример: в каждом системном вызове сохранять младшие биты из счетчика аппаратных циклов и комбинировать их с помощью операции ИСКЛЮЧАЮЩЕЕ ИЛИ (XOR) с системным *пулом энтропии*. Генератор PRNG размещается глубоко в недрах ядра и периодически повторно инициализируется битами из пула энтропии; выходная последовательность этого PRNG передается прикладным программам. В Linux низкоуровневый пул энтропии доступен как `/dev/random`, а выходная последовательность генератора PRNG – как `/dev/urandom`. К счастью, нет никакой необходимости напрямую обращаться к этим устройствам; стандартная библиотека C++ уже позаботилась об этом.

Проблема функции rand()

В C для получения случайных чисел издавна используется функция `rand()`. Она присутствует также в стандартной библиотеке C++, не принимает аргументов и возвращает единственное целое число из диапазона `[0, RAND_MAX]`, подчиняющееся закону равномерного распределения. Внутреннее состояние можно инициализировать вызовом библиотечной функции `srand(seed_value)`.

Классический способ получения случайных чисел в диапазоне `[0, x)` не изменился с 1980-х годов:

```
#include <stdlib.h>

int randint0(int x) {
  return rand() % x;
}
```

Однако такой подход имеет несколько проблем. Первая и самая очевидная – генерируемые значения x имеют неодинаковую вероятность. Допустим, ради полемики, что `rand()` возвращает равномерно распределенные значения из диапазона [0, 32767], тогда `randint0(10)` будет возвращать значения из диапазона [0, 7] на 1/ 3276 чаще, чем 8 или 9.

Вторая проблема: `rand()` использует глобальное состояние; все потоки выполнения в программе на C++ будут использовать один и тот же генератор случайных чисел. Это не порождает проблемы безопасности в многопоточной среде – начиная с C++11 `rand()` гарантирует такую безопасность. Но это проблема производительности (потому что каждый вызов `rand()` запирает глобальный мьютекс) и воспроизводимости (потому что если `rand()` используется несколькими потоками конкурентно, в разных сеансах выполнения программы будут производиться разные результаты).

Третья проблема тесно связана с глобальным характером состояния `rand()` – любая функция в программе может изменить это состояние простым вызовом `rand()`. Это делает невозможным использовать `rand()` в модульных тестах. Взгляните на следующий фрагмент:

```
int heads(int n) {
  DEBUG_LOG("heads");
  int result = 0;
  for (int i = 0; i < n; ++i) {
    result += (rand() % 2);
  }
  return result;
}

void test_heads() {
  srand(17); // зафиксировать начальное значение
  int result = heads(42);
  assert(result == 27);
}
```

Очевидно, что работа модульного теста `test_heads` нарушится, если мы начнем распараллеливать модульные тесты (потому что вызовы `rand()` в некотором другом потоке выполнения будут мешать точности вычислений в этом тесте). Более того, его работа может также нарушиться, если кто-то изменит реализацию `DEBUG_LOG` и добавит или удалит вызов `rand()`! Подобные проблемы часто возникают, когда архитектура программы зависит от глобальных переменных. Мы наблюдали аналогичную опасность с `std::pmr::get_default_resource()` в главе 8 «Диспетчеры памяти». Во всех таких случаях я на-

стоятельно советую одно и то же средство защиты – *не использовать глобальных переменных, не использовать глобального состояния*.

Итак, библиотека C страдает двумя проблемами – она не обеспечивает по-настоящему равномерное распределение псевдослучайных чисел и зависит от глобальных переменных. Давайте посмотрим, как заголовок `<random>` из стандартной библиотеки C++ решает обе эти проблемы.

Решение проблем с <random>

Заголовок `<random>` определяет два основных понятия – *генератор* (generator) и *распределение* (distribution). *Генератор* (класс, моделирующий понятие `UniformRandomBitGenerator`) инкапсулирует внутреннее состояние PRNG в объект C++ и предоставляет функцию-член для получения следующего значения в форме оператора вызова функции `operator()(void)`. *Распределение* (класс, моделирующий понятие `RandomNumberDistribution`) – это своеобразный фильтр, через который пропускается вывод генератора, чтобы вместо равномерно распределенных случайных битов, которые возвращает `rand()`, можно было получать значения, распределенные согласно заданному математическому закону, и ограничить их определенным диапазоном, как `rand() % n`, но математически более точно и в целом более гибко.

Заголовок `<random>` определяет семь типов *генераторов* и двенадцать типов *распределений*. Многие из них являются шаблонами со множеством параметров. Большинство генераторов представляет скорее исторический интерес, чем практический, и подавляющее количество распределений интересны только математикам. Поэтому в этой главе мы сосредоточимся лишь на нескольких стандартных генераторах и распределениях, иллюстрирующих интересные стороны стандартной библиотеки.

Генераторы

Любой объект *генератора* `g` поддерживает следующие операции:

- `g()`: перекодирует внутреннее состояние генератора и возвращает следующее значение;
- `g.min()`: возвращает наименьшее значение, которое может вернуть `g()` (обычно 0);
- `g.max()`: возвращает наибольшее значение, которое может вернуть `g()`, то есть диапазон значений, которые может вернуть `g()`, простирается от `g.min()` до `g.max()` включительно;
- `g.discard(n)`: фактически выполняет `n` вызовов `g()` и отбрасывает результаты. В библиотеке с хорошей реализацией вы заплатите временем, необходимым на `n` изменений внутреннего состояния генератора, но сэкономите на вычислениях, связанных с получением следующих значений из этого состояния.

Истинно случайные биты и std::random_device

`std::random_device` – это *генератор*. Он имеет очень простой интерфейс; это даже не шаблонный класс, а самый простой традиционный класс. После создания экземпляра `std::random_device` с помощью конструктора по умолчанию можно сразу же использовать его перегруженный оператор вызова для получения значений типа `unsigned int`, равномерно распределенных в диапазоне `[rd.min(), rd.max()]`.

Будьте внимательны – `std::random_device` не полностью моделирует понятие `UniformRandomBitGenerator`. Самое важное: он не может копироваться или перемещаться. Однако на практике это не является проблемой, потому что обычно не возникает необходимости долго хранить генератор истинно случайных чисел. Чаще используются короткоживущие экземпляры `std::random_device`, чтобы сгенерировать начальное значение для долгоживущего генератора псевдослучайных чисел некоторого другого типа, например:

```
std::random_device rd;
unsigned int seed = rd();
assert(rd.min() <= seed && seed <= rd.max());
```

Теперь перейдем к единственному генератору псевдослучайных чисел, знать который вам действительно необходимо.

Псевдослучайные биты с std::mt19937

Единственный генератор псевдослучайных чисел, о котором вам действительно нужно знать, называется *Mersenne Twister*. Этот алгоритм появился в 1997 году, и его высококачественные реализации можно найти в любом языке программирования. Технически алгоритм Mersenne Twister определяет целое семейство родственных генераторов PRNG – алгоритмически он подобен шаблонам в C++, – но в практике чаще других членов этого семейства используется **MT19937**. Эта строка цифр похожа на дату, но в действительности это не так; это размер внутреннего состояния алгоритма в битах. Поскольку функция алгоритма Mersenne Twister, возвращающая следующее значение, изменяет это состояние, она фактически переберет все возможные варианты состояния, прежде чем вернется к началу цикла – период генератора MT19937 равен $2^{19937} - 1$. Сравните это с нашей реализацией `SimplePRNG` в начале главы, которая имеет 32-битное внутреннее состояние и период 2^{31}. (Наш генератор `SimplePRNG` имеет 2^{32} возможных внутренних состояния, но только половина из них будет достигнута, прежде чем цикл начнется снова. Например, `state=3` недостижимо при начальном значении `state=1`.)

Но хватит теории. Давайте рассмотрим Mersenne Twister в действии! *Шаблонному алгоритму* Mersenne Twister в языке C++ соответствует шаблонный класс `std::mersenne_twister_engine<...>`, но вам не нужно использовать его непосредственно; вы должны использовать вспомогательный тип `std::mt19937`, как показано ниже:

```
std::mt19937 g;
assert(g.min() == 0 && g.max() == 4294967295);

assert(g() == 3499211612);
assert(g() == 581869302);
assert(g() == 3890346734);
```

Конструктор по умолчанию для `std::mt19937` инициализирует внутреннее состояние известным стандартным значением. Это гарантирует, что выходная последовательность, генерируемая объектом `mt19937`, созданным конструктором по умолчанию, будет идентична на всех платформах – в отличие от `rand()`, которая на разных платформах часто генерирует разные последовательности.

Чтобы получить другую выходную последовательность, нужно передать *начальное значение* конструктору `std::mt19937`. Сделать это в C++17 можно двумя способами – утомительным и простым. Утомительный способ состоит в конструировании истинно случайного 19937-битного начального значения и его копировании в объект `std::mt19937` в виде *начальной последовательности*, как показано ниже:

```
std::random_device rd;

uint32_t numbers[624];
std::generate(numbers, std::end(numbers), std::ref(rd));
  // Сгенерировать начальное состояние.

SeedSeq sseq(numbers, std::end(numbers));
  // Скопировать созданное состояние в "начальную последовательность".

std::mt19937 g(sseq);
  // Инициализировать генератор mt19937 скопированным состоянием.
```

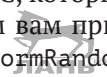
Здесь тип `SeedSeq` может быть `std::seed_seq` (в действительности `std::vector`, использует память в куче) или вручную написанным классом «начальной последовательности», как в следующем примере:

```
template<class It>
struct SeedSeq {
  It begin_;
  It end_;
public:
  SeedSeq(It begin, It end) : begin_(begin), end_(end) {}

  template<class It2>
  void generate(It2 b, It2 e) {
    assert((e - b) <= (end_ - begin_));
    std::copy(begin_, begin_ + (e - b), b);
  }
};
```

Как видите, для создания единственного объекта PRNG приходится писать много кода! (Я предупреждал, что это утомительный способ.) Простой способ, который обычно используется в практике, заключается в инициализации MT19937 простым 32-битным целым числом:

```
std::random_device rd;

std::mt19937 g(rd());
  // 32 случайных бит достаточно для любых применений!
  // ...Верно?
```

Внимание! Число 32 намного меньше 19937! Этот упрощенный способ инициализации позволяет произвести только четыре миллиарда выходных последовательностей. То есть, запуская программу снова и снова со случайными начальными значениями, можно заметить некоторую повторяемость после нескольких десятков тысяч прогонов. (Это проявление знаменитого *парадокса дней рождений*.) Кроме того, если степень предсказуемости важна для вас, вы должны также понимать, что Mersenne Twister не является криптографически безопасным. То есть даже если вы инициализируете его истинно случайным 19937-битным начальным значением, злоумышленник сможет перебрать все варианты 19937 бит и точно предсказать все последующие результаты, имея всего лишь несколько сотен членов вашей выходной последовательности. Если вам требуется **криптографически безопасный генератор псевдослучайных чисел** (Cryptographically Secure Pseudo-Random Number Generator, CSPRNG), используйте такие алгоритмы, как AES-CTR или ISAAC, которые, однако, не реализуются стандартной библиотекой C++. При этом вам придется завернуть реализацию CSPRNG в класс, моделирующий `UniformRandomBitGenerator`, чтобы его можно было использовать в стандартных алгоритмах, о которых рассказывается ближе к концу этой главы.



Фильтрация вывода генераторов с помощью адаптеров

Выше уже упоминалось, что вывод *генератора* обычно фильтруется с помощью распределения, чтобы преобразовать биты, возвращаемые генератором, в значения, пригодные к использованию. Самое интересное, что вывод генератора можно направить в *адаптер генератора*, который может преобразовать биты в разные полезные значения. В стандартной библиотеке реализовано три адаптера: `std::discard_block_engine`, `std::shuffle_order_engine` и `std::independent_bits_engine`. Эти адаптеры действуют подобно *адаптерам контейнеров* (таким как `std::stack`), обсуждавшимся в главе 4 «Зоопарк контейнеров», – они предоставляют некоторый интерфейс, но большинство деталей реализации делегируют другим классам.

Экземпляр `std::discard_block_engine<Gen, p, r>` хранит *внутренний генератор* типа `Gen` и делегирует ему реализацию всех своих операций, кроме того, `discard_block_engine::operator()` возвращает только первые `r` из каждых `p` выводов внутреннего генератора. Например:

```
std::vector<uint32_t> raw(10), filtered(10);

std::discard_block_engine<std::mt19937, 3, 2> g2;
std::mt19937 g1 = g2.base();

std::generate(raw.begin(), raw.end(), g1);
std::generate(filtered.begin(), filtered.end(), g2);

assert(raw[0] == filtered[0]);
assert(raw[1] == filtered[1]);
  // raw[2] не появится в filtered[]

assert(raw[3] == filtered[2]);
assert(raw[4] == filtered[3]);
  // raw[5] не появится в filtered[]
```

Примечательно, что ссылку на внутренний генератор можно получить вызовом `g2.base()`. В предыдущем примере `g1` инициализируется как копия `g2.base()`; это объясняет, почему вызов `g1()` не влияет на состояние `g2`, и наоборот.

Экземпляр `std::shuffle_order_engine<Gen, k>` хранит буфер с последними *k* выводами внутреннего генератора и дополнительное целое число `Y`. Каждый вызов `shuffle_order_engine::operator()` вычисляет `Y = buffer[Y % k]` и затем записывает `buffer[Y] = base()()`. (В действительности формула вычисления индекса `Y` в буфере намного сложнее, но суть ее та же.) Примечательно, что `std::shuffle_order_engine` не использует `std::uniform_int_distribution` для отображения `Y` в диапазон `[0, k)`. Это не влияет на *случайность* значений, возвращаемых генератором, – если внутренний генератор уже псевдослучайный, перемешивание вывода не сделает его более или менее случайным, какой бы алгоритм ни использовался для перемешивания. То есть алгоритм, используемый в `shuffle_order_engine`, был выбран, скорее, по историческим причинам – он служит строительным блоком классического алгоритма, описанного в книге Дональда Кнута (Donald Knuth) «Искусство программирования»:

```
using knuth_b = std::shuffle_order_engine<
  std::linear_congruential_engine<
    uint_fast32_t, 16807, 0, 2147483647
  >,
  256
>;
```

Экземпляр `std::independent_bits_engine<Gen, w, T>` не хранит никакого другого состояния, кроме внутреннего генератора типа `Gen`. Функция `independent_bits_engine::operator()` вызывает `base()()` столько раз, сколько необходимо для вычисления не менее `w` случайных битов; затем объединяет `w` из этих битов (используя алгоритм, который имеет скорее исторический интерес, чем практический) и интерпретирует их как целое без знака типа `T`. (Если `T`

не является целочисленным беззнаковым типом или содержит меньше битов, чем w, это считается ошибкой.)

Вот пример объединения битов из нескольких вызовов base()() в independent_bits_engine:

```
std::independent_bits_engine<std::mt19937, 40, uint64_t> g2;
std::mt19937 g1 = g2.base();

assert(g1() == 0xd09'1bb5c);   // Получить "1bb5c"...
assert(g1() == 0x22a'e9ef6);   // и "e9ef6"...
assert(g2() == 0x1bb5c'e9ef6); // Объединить и передать!
```

А вот пример использования independent_bits_engine для отсечения всех, кроме значимых цифр из вывода mt19937 (создание генератора *броска монеты*) с последующим объединением 32 выводов этого генератора для сборки 32-битного генератора:

```
using coinflipper = std::independent_bits_engine<
  std::mt19937, 1, uint8_t>;

coinflipper onecoin;
std::array<int, 64> results;
std::generate(results.begin(), results.end(), onecoin);
assert((results == std::array<int, 64>{{
  0,0,0,1, 0,1,1,1, 0,1,1,1, 0,0,1,0,
  1,0,1,0, 1,1,1,1, 0,0,0,1, 0,1,0,1,
  1,0,0,1, 1,1,1,0, 0,0,1,0, 1,0,1,0,
  1,0,0,1, 0,0,0,0, 0,1,0,0, 1,1,0,0,
}}));
```


```
std::independent_bits_engine<coinflipper, 32, uint32_t> manycoins;
assert(manycoins() == 0x1772af15);
assert(manycoins() == 0x9e2a904c);
```

Обратите внимание, что independent_bits_engine не выполняет никаких сложных операций с битами из внутреннего генератора; в частности, он предполагает, что внутренний генератор действует непредвзято. Если генератор WeightedCoin будет отдавать предпочтение четным числам, это смещение будет также наблюдаться в выводе independent_bits_engine<WeightedCoin, w, T>.

Несмотря на то что мы потратили несколько страниц на обсуждение генераторов, у вас нет никаких причин использовать эти малопонятные классы в своем коде! Если вам нужен PRNG, используйте std::mt19937; если вам нужен криптографически безопасный PRNG, используйте такие алгоритмы, как AES-CTR или ISAAC; а если вам нужно небольшое количество по-настоящему случайных битов, чтобы инициализировать свой PRNG, используйте std::random_device. Это все генераторы, которые вам доведется применять на практике.

Распределения

Теперь, узнав, как генерируются случайные биты, посмотрим, как преобразовать эти случайные биты в числовые значения, подчиняющиеся определенному закону распределения. Этот процесс из двух этапов – генерация битов и их последующее преобразование в значения данных – очень похож на процесс буферизации и парсинга, рассматривавшийся в главе 9 «Потоки ввода/вывода». Сначала нужно получить биты и байты, а затем выполнить некоторую операцию для их преобразования в значения определенных типов.

Любой объект распределения `dist` позволяет выполнять следующие операции с ним.

- `dist(g)`: возвращает следующий результат, соответствующий заданному математическому распределению. Для этого может потребоваться выполнить несколько вызовов `g()` или ни одного, в зависимости от внутреннего состояния объекта `dist`.
- `dist.reset()`: очищает внутреннее состояние объекта `dist`, если имеется. Вам никогда не придется использовать эту функцию-член.
- `dist.min()` и `dist.max()`: возвращают наименьшее и наибольшее значения, которое можно получить вызовом `dist(g)` для любого генератора `g`. Обычно эти числа либо самоочевидны, либо бессмысленны; например, `std::normal_distribution<float>().max()` вернет `INFINITY`.

Посмотрим, как действуют распределения.

Имитация броска игровой кости с uniform_int_distribution

Метод `std::uniform_int_distribution` – самый простой тип распределения в стандартной библиотеке. Он выполняет те же операции, которые мы пытались выполнять с `randint0` выше в этой главе, – отображение случайного целого числа в заданный диапазон, – но делает это без какого-либо смещения. Простейшая реализация `uniform_int_distribution` выглядит примерно так:

```
template<class Int>
class uniform_int_distribution {
  using UInt = std::make_unsigned_t<Int>;
  UInt m_min, m_max;
public:
  uniform_int_distribution(Int a, Int b) :
    m_min(a), m_max(b) {}

  template<class Gen>
  Int operator()(Gen& g) {
    UInt range = (m_max - m_min);
    assert(g.max() - g.min() >= range);
    while (true) {
      UInt r = g() - g.min();
```

```
        if (r <= range) {
          return Int(m_min + r);
        }
      }
    }
  };
```

Фактическая реализация в стандартной библиотеке должна сделать нечто, чтобы избавиться от проверки `assert`. Как правило, с этой целью используется что-то вроде `independent_bits_engine`, чтобы получить точно `ceil(log2(range))` случайных битов, минимизируя количество циклов `while`.

Как подразумевает предыдущий пример, `uniform_int_distribution` не имеет состояния (хотя *технически* это не гарантируется) и поэтому чаще других используется для создания нового объекта распределения, когда требуется сгенерировать число. То есть нашу функцию `randint0` можно реализовать, как показано ниже:

```
int randint0(int x) {
  static std::mt19937 g;
  return std::uniform_int_distribution<int>(0, x-1)(g);
}
```

Теперь самое время отметить некоторые странности механизмов из `<random>`. Как правило, когда вы определяете *целочисленный диапазон* для одной из этих функций или конструкторов, он интерпретируется как *закрытый*. Это резко контрастирует с привычной интерпретацией диапазонов в C и C++; мы даже говорили в главе 3 «Алгоритмы с парами итераторов», что отклонение от правила полуоткрытых диапазонов обычно служит признаком ошибочного кода. Но в случае с механизмами получения случайных чисел действует иное правило – диапазоны интерпретируются как закрытые. Почему?

Важнейшее преимущество полуоткрытых диапазонов заключается в простоте представления *пустого диапазона*. С другой стороны, полуоткрытые диапазоны неспособны представлять *полные диапазоны*, то есть диапазоны, охватывающие всю область значений. (В главе 4 «Зоопарк контейнеров» мы видели, какие сложности возникают из-за этого в реализации `std::list<T>::end()`.) Предположим, что нам нужно выразить идею равномерного распределения по всему диапазону значений типа `long long`. У нас не получится сделать это с применением полуоткрытого диапазона `[LLONG_MIN, LLONG_MAX+1)`, потому что `LLONG_MAX+1` вызовет переполнение. Но получится с использованием закрытого диапазона `[LLONG_MIN, LLONG_MAX]` – именно так поступают функции и классы из библиотеки `<random>` (такие как `uniform_int_distribution`). Метод `uniform_int_distribution<int>(0,6)` – это распределение по диапазону, включающему семь чисел `[0,6]`, а `uniform_int_distribution<int>(42,42)` – допустимое распределение, которое всегда возвращает 42.

С другой стороны, `std::uniform_real_distribution<double>(a, b)` работает с *полуоткрытым* диапазоном! Метод `std::uniform_real_distri-`

bution<double>(0, 1) возвращает значения типа double, равномерно распределенные в диапазоне [0, 1). В мире вещественных чисел не существует проблемы переполнения – фактически есть возможность выразить полуоткрытый диапазон [0, INFINITY), хотя, конечно же, не существует такого понятия, как *равномерное распределение в бесконечном диапазоне*. Кроме того, при использовании вещественных чисел стирается грань между полуоткрытыми и закрытыми диапазонами; например, std::uniform_real_distribution<float>(0, 1)(g) на вполне законных основаниях может вернуть float(1.0), если будет сгенерировано число, достаточно близкое к 1, из-за округления вверх примерно каждого 225-го результата. (На момент написания этих строк библиотека libc++ действовала, как описано здесь. Для GNU libstdc++ была выпущена «заплата», которая округляла вниз числа, близкие к 1, чтобы результат меньше 1.0 появлялся чаще, чем можно было бы предсказать.)

Генерирование выборок с normal_distribution

Наиболее практичным, пожалуй, является *нормальное распределение*, также известное как **колоколообразная кривая**. В реальном мире нормальное распределение встречается повсюду, в частности в распределении физических признаков в популяции. Например, гистограмма роста взрослых людей имеет вид кривой нормального распределения – больше всего людей сосредоточено около среднего значения, и их число уменьшается к краям. Это означает, что нормальное распределение можно использовать для присваивания веса, роста и других признаков персонажам в игре.

Метод std::normal_distribution<double>(m, sd) конструирует экземпляр normal_distribution<double> с заданным средним (m) и стандартным отклонениями (sd). (По умолчанию эти параметры получают значения m=0 и sd=1, так что внимательно следите за опечатками!) Вот пример использования normal_distribution для создания выборки из 10 000 нормально распределенных образцов с последующей проверкой параметров их распределения:

```
double mean = 161.8;
double stddev = 6.8;
std::normal_distribution<double> dist(mean, stddev);

  // Инициализировать генератор.
std::mt19937 g(std::random_device{}());

  // Заполнить вектор 10 000 образцов.
std::vector<double> v;
for (int i=0; i < 10000; ++i) {
  v.push_back( dist(g) );
}
std::sort(v.begin(), v.end());

  // Сравнить ожидания с реальностью.
```

```
auto square = [](auto x) { return x*x; };
double mean_of_values = std::accumulate(
  v.begin(), v.end(), 0.0) / v.size();
double mean_of_squares = std::inner_product(
  v.begin(), v.end(), v.begin(), 0.0) / v.size();
double actual_stddev =
  std::sqrt(mean_of_squares - square(mean_of_values));
printf("Expected mean and stddev: %g, %g\n", mean, stddev);
printf("Actual mean and stddev: %g, %g\n",
       mean_of_values, actual_stddev);
```

В отличие от других распределений, которые мы увидим в этой главе, `std::normal_distribution` имеет внутреннее состояние. Для каждого генерируемого значения (образца) можно создать новый экземпляр `std::normal_distribution`, но это нанесет ущерб эффективности работы программы. Это объясняется тем, что наиболее популярный алгоритм нормального распределения значений производит два независимых значения в каждом шаге; `std::normal_distribution` не может вернуть сразу два значения, поэтому одно из них запоминается в переменной-члене и возвращается вам при следующем обращении. Для очистки хранимого состояния можно использовать функцию-член `dist.reset()`, но вам едва ли это потребуется.

Взвешенный выбор с discrete_distribution

Метод `std::discrete_distribution<int>(wbegin, wend)` конструирует дискретное, или взвешенное, распределение с полуоткрытым диапазоном `[0, wend - wbegin)`. Принцип его действия проще объяснить на примере:

```
template<class Values, class Weights, class Gen>
auto weighted_choice(const Values& v, const Weights& w, Gen& g)
{
  auto dist = std::discrete_distribution<int>(
    std::begin(w), std::end(w));
  int index = dist(g);
  return v[index];
}

void test() {
  auto g = std::mt19937(std::random_device{}());
  std::vector<std::string> choices =
    { "quick", "brown", "fox" };

  std::vector<int> weights = { 1, 7, 2 };
  std::string word = weighted_choice(choices, weights, g);
    // в 7 случаях из 10 ожидается, что word=="brown".
}
```

Метод `std::discrete_distribution<int>` создает внутреннюю копию `weights` в скрытой переменной-члене типа `std::vector<double>` (и, как обыч-

но для <random>, не поддерживает выбор диспетчера памяти). Копию этого вектора можно получить вызовом dist.probabilities():

```
int w[] = { 1, 0, 2, 1 };
std::discrete_distribution<int> dist(w, w+4);
std::vector<double> v = dist.probabilities();
assert((v == std::vector{ 0.25, 0.0, 0.50, 0.25 }));
```

Вам редко придется использовать discrete_distribution непосредственно; в лучшем случае вы инкапсулируете его в нечто вроде предыдущей функции weighted_choice. Чтобы избежать использования динамической памяти или операций с вещественными числами, можно написать более простую функцию, не использующую динамическую память, как показано ниже:

```
template<class Values, class Gen>
auto weighted_choice(
  const Values& v, const std::vector<int>& w,
  Gen& g)
{
  int sum = std::accumulate(w.begin(), w.end(), 0);
  int cutoff = std::uniform_int_distribution<int>(0, sum - 1)(g);
  auto vi = v.begin();
  auto wi = w.begin();
  while (cutoff > *wi) {
    cutoff -= *wi++;
    ++vi;
  }
  return *vi;
}
```

Однако выбор вещественной арифметики для библиотечной реализации *по умолчанию* discrete_distribution имеет веское основание: она избавляет от беспокойства о целочисленном переполнении. Предыдущий код начнет производить ошибочные результаты, если значение суммы превысит диапазон int.

Перемешивание карт с std::shuffle

Завершим эту главу знакомством с std::shuffle(a,b,g), стандартным алгоритмом, который принимает генератор случайных чисел. Это перестановочный алгоритм, как определено в главе 3 «Алгоритмы с парами итераторов». Он принимает диапазон элементов [a,b) и перемешивает их, сохраняя значения, но не местоположения.

Метод std::shuffle(a,b,g) появился в C++11 взамен более старого алгоритма std::random_shuffle(a,b). Старый алгоритм «случайно» перемешивал диапазон [a,b), но ему нельзя было передать источник случайных чисел. На практике это означало использование глобальной функции rand() со всеми сопутствующими проблемами. Как только стандарт C++11 добавил заголовок

<random>, определяющий стандартный способ представления генераторов случайных чисел, появилась возможность избавиться от старого алгоритма random_shuffle, основанного на rand(), и начиная с C++17 std::random_shuffle(a,b) больше не является частью стандартной библиотеки C++.

Вот как можно использовать C++11-версию std::shuffle для перемешивания колоды игральных карт:

```
std::vector<int> deck(52);
std::iota(deck.begin(), deck.end(), 1);
  // теперь deck содержит целые числа от 1 до 52.

std::mt19937 g(std::random_device{}());
std::shuffle(deck.begin(), deck.end(), g);
  // теперь содержимое deck перемешано в случайном порядке.
```

Напомню, что все *генераторы* в <random> полностью детерминированы, то есть экземпляр std::mt19937, например, инициализированный фиксированным значением, будет производить в точности одну и ту же последовательность на любой платформе. Но это не относится к распределениям, таким как uniform_real_distribution, и не относится к алгоритму shuffle. Переключение с libc++ на libstdc++ или даже простое обновление версии компилятора может изменить поведение std::shuffle.

Обратите внимание, что в предыдущем примере используется «простой» метод инициализации алгоритма Mersenne Twister, а это значит, что он будет поддерживать только 4×10^9 разных вариантов перемешивания из 8×10^{67} возможных! Если вы собираетесь организовать перемешивание карт для игры в настоящем казино, вам определенно стоит использовать «утомительный» метод инициализации, описанный выше в этой главе, или – еще проще, если производительность не является проблемой, – используйте std::random_device непосредственно:

```
std::random_device rd;
std::shuffle(deck.begin(), deck.end(), rd);
  // теперь содержимое deck перемешано в ИСТИННО случайном порядке.
```

Вы можете использовать в std::shuffle любой генератор и метод инициализации. В этом заключается огромное преимущество комбинируемого подхода к генерации случайных чисел, реализованного в стандартной библиотеке.

Итоги

Стандартная библиотека реализует два механизма, имеющих отношения к генерации случайных чисел, – *генераторы* и *распределения*. Генераторы имеют внутреннее состояние, должны инициализироваться начальным значением и производят беззнаковые целочисленные значения (биты) посредством operator()(void). К наиболее важным относятся два типа генераторов:

`std::random_device`, который производит по-настоящему случайные биты, и `std::mt19937`, который производит псевдослучайные биты.

Распределения *обычно* не имеют своего состояния и производят числовые значения посредством `operator()(Gen&)`. Чаще других в практике программирования используется распределение `std::uniform_int_distribution<int>(a,b)`, которое производит целые числа в закрытом диапазоне `[a,b]`. В стандартной библиотеке реализованы также другие распределения, такие как `std::uniform_real_distribution`, `std::normal_distribution` и `std::discrete_distribution`, а также несколько довольно замысловатых распределений, более интересных математикам и статистикам.

Примером использования случайности может служить стандартный алгоритм `std::shuffle`, который заменил более старый `std::random_shuffle`. Не используйте `random_shuffle` в новом коде.

Помните, что `std::mt19937` действует одинаково на всех платформах, но это не относится к распределениям и `std::shuffle`.

Глава 12

Файловая система

Одним из самых заметных новшеств, появившихся в C++17, стала библиотека <filesystem>. Эта библиотека, как и многие другие важные особенности современного C++, была заимствована из проекта Boost. В 2015 г. она была включена в техническую спецификацию стандарта для сбора отзывов и, наконец, вошла в состав стандарта C++17 с некоторыми изменениями, внесенными по результатам отзывов.

В этой главе вы познакомитесь:

- с реализацией в <filesystem> возврата динамически типизируемых ошибок без возбуждения исключений, и как то же самое можно реализовать вручную;
- с форматом *пути* и принципиально не совместимыми положениями POSIX и Windows;
- с приемами получения информации о файлах и обхода каталогов на основе использования переносимых алгоритмов из C++17;
- со способами создания, копирования, переименования и удаления файлов и каталогов;
- с методом определения объема свободного пространства в файловой системе.

Примечание о пространствах имен

Все стандартные инструменты для работы с файловыми системами в C++17 сосредоточены в единственном заголовке, <filesystem>, и в единственном пространстве имен `namespace std::filesystem`. Такая организация следует прецеденту, созданному заголовком <chrono> в C++11 с его пространством имен `std::chrono`. (В этой книге отсутствует полное описание <chrono>. А использование соответствующих инструментов с `std::thread` и `std::timed_mutex` кратко описывается в главе 7 «Конкуренция».)

Такая организация пространств имен означает, что при использовании инструментов из <filesystem> требуется использовать такие идентификаторы, как `std::filesystem::directory_iterator` и `std::filesystem::temp_directory_path()`. Эти полные имена выглядят довольно громоздко! Но внедрение всего пространства имен в текущий контекст с помощью объявления `using`, –

вероятно, еще худший выбор. Последнее десятилетие нас настойчиво учили не использовать объявление `using namespace std`, и этот совет распространяется на все пространства имен в стандартной библиотеке, как бы глубоко они не были вложены. Взгляните на следующий код:

```
using namespace std::filesystem;

void foo(path p)
{
  remove(p); // Какая функция вызывается здесь?
}
```

Гораздо лучше определить *псевдоним пространства имен* в области видимости файла (в файле `.cc`) или в области видимости пространства имен (в файле `.h`). Псевдоним позволит ссылаться на пространство имен по новому имени, как показано ниже:


```
namespace fs = std::filesystem;

void foo(fs::path p)
{
  fs::remove(p); // Намного понятнее!
}
```

Для ссылки на пространство имен `std::filesystem` в оставшейся части этой главы я буду использовать псевдоним `fs`. Под ссылкой `fs::path` я буду подразумевать `std::filesystem::path`. А под ссылкой `fs::remove` я буду подразумевать `std::filesystem::remove`.

Определение глобального псевдонима пространства имен `fs` где-то еще имеет также чисто прагматическое преимущество. На момент написания этих строк все производители реализаций стандартной библиотеки, кроме Microsoft Visual Studio, заявили о добавлении заголовка `<filesystem>`. Однако инструменты в `<filesystem>` очень похожи на инструменты в libstdc++ и libc++ из заголовка `<experimental/filesystem>` и в Boost из `<boost/filesystem.hpp>`. Поэтому, если последовательно ссылаться на них с использованием псевдонима пространства имен, такого как `fs`, вы сможете переключаться с одной реализации на другую, просто изменяя определение псевдонима, вместо того чтобы выполнять поиск с заменой во всей базе кода. Следующий фрагмент иллюстрирует правоту моих слов:

```
#if USE_CXX17
  #include <filesystem>
  namespace fs = std::filesystem;
#elif USE_FILESYSTEM_TS
  #include <experimental/filesystem>
  namespace fs = std::experimental::filesystem;
#elif USE_BOOST
  #include <boost/filesystem.hpp>
```

```
  namespace fs = boost::filesystem;
#endif
```

Очень длинное примечание об уведомлениях об ошибках

Программисты на C++ испытывают любовь и ненависть к уведомлениям об ошибках. В данном случае под «уведомлениями об ошибках» я подразумеваю: «что делать, если нет возможности выполнить запрошенную операцию». Классический, типичный и все еще лучший вариант уведомить о проблеме в C++ – возбудить исключение. В предыдущих главах мы видели, что иногда исключение – единственный разумный способ из-за невозможности вернуть что-то вызывающему коду. Например, если вашей задачей было конструирование объекта, и операция его создания потерпела неудачу, вы ничего не сможете вернуть; когда конструктор терпит неудачу, остается единственный путь – возбудить исключение. Но мы также видели, что (в главе 9 «Потоки ввода/вывода»), что библиотека `<iostream>` отступает от этого правила! Если операция создания объекта `std::fstream` потерпела неудачу (например, из-за невозможности открыть указанный файл), она не возбудит исключения; вместо этого вы получите полноценный объект `fstream` с `f.fail() && !f.is_open()`.

Причина такого «плохого» поведения `fstream` заключается в относительно высокой вероятности, что указанный файл не может быть открыт. Возбуждение исключения каждый раз, когда нельзя открыть файл, по своему неудобству близко к использованию исключений для управления порядком выполнения, и этого желательно избегать. Поэтому, чтобы не заставлять программиста повсеместно использовать конструкцию `try/catch`, библиотека возвращает тот же результат, что и при успешной операции, но дает возможность проверить результат (с помощью обычной инструкции `if`, а не `catch`).

Благодаря этому можно избежать такого запутанного кода:

```
try {
  f.open("hello.txt");
  // Файл благополучно открыт.
} catch (const std::ios_base::failure&) {
  // При открытии произошла ошибка.
}
и записать то же самое проще:
f.open("hello.txt");
if (f.is_open()) {
  // Файл благополучно открыт.
} else {
  // При открытии произошла ошибка.
}
```

Этот подход хорошо работает там, где результатом операции является тяжеловесный объект (например, `fstream` в iostreams), имеющий признак *ошибочно-*

го состояния, или где такой признак можно добавить на этапе проектирования. Но он имеет свои недостатки и неприменим в ситуациях, где нет такого тяжеловесного типа. Мы видели подобную ситуацию в главе 9 «Потоки ввода/вывода», когда рассматривали способы парсинга целых чисел из строк. Если ошибка маловероятна или «использование исключений для управления порядком выполнения» не ухудшает производительность, мы используем `std::stoi`:

```
// Подход на основе исключений.
try {
  int i = std::stoi(s);
  // Парсинг выполнен успешно.
} catch (...) {
  // Ошибка парсинга.
}
```

А если нужна совместимость с C++03, мы используем `strtol`, которая уведомляет об ошибке через глобальную переменную (локальную для потоков выполнения) переменную `errno`:

```
char *endptr = nullptr;
errno = 0;
long i = strtol(s, &endptr, 10);
if (endptr != s && !errno) {
  // Парсинг выполнен успешно.
} else {
  // Ошибка парсинга.
}
```

Следуя суперсовременному стилю C++17, мы используем `std::from_chars`, которая возвращает легковесную структуру с указателем на конец строки и значением типа перечисления `std::errc`, определяющим успех или неудачу:

```
int i = 0;
auto [ptr, ec] = std::from_chars(s, end(s), i);
if (ec != std::errc{}) {
  // Парсинг выполнен успешно.
} else {
  // Ошибка парсинга.
}
```

Библиотеке `<filesystem>` требуется аналогичная возможность уведомления об ошибках, как в `std::from_chars`. Практически любая операция с файловой системой может завершиться неудачей из-за действий других процессов, выполняющихся в системе; поэтому возбуждение исключений (а-ля `std::stoi`) по своему неудобству близко к использованию исключений для управления порядком выполнения. Но передача и проверка «ошибочного результата», такого как `ec`, тоже может стать утомительным занятием, чреватым ошибками. Поэтому было решено реализовать по два интерфейса почти для каждой функции в заголовке `<filesystem>`!

Например, вот две функции из <filesystem>, определяющие размер файла на диске:

```
uintmax_t file_size(const fs::path& p);

uintmax_t file_size(const fs::path& p,
  std::error_code& ec) noexcept;
```

Обе принимают fs::path (обсуждается далее в этой главе) и возвращают значение uintmax_t с размером указанного файла в байтах. Но что случится, если указанный файл не существует или существует, но текущий пользователь не имеет права запрашивать его размер? Первая функция в этом случае *возбудит исключение* fs::filesystem_error, а вторая (которая фактически отмечена спецификатором noexcept) заполнит параметр типа std::error_code информацией об ошибке (или очистит его, если операция выполнилась успешно).

Сравнив сигнатуры fs::file_size и std::from_chars, можно заметить, что from_chars принимает параметр типа std::errc, а file_size – параметр типа std::error_code. Эти два типа, несмотря на сходство, далеко не одно и то же! Чтобы понять разницу – и саму архитектуру <filesystem> API, не возбуждающего исключений, – совершим короткий тур по другой части стандартной библиотеки C++11.

Использование <system_error>

Различия между механизмами уведомлений об ошибках в std::from_chars и fs::file_size обусловлены их природной сложностью. from_chars может потерпеть неудачу в двух случаях – либо когда строка не начинается с цифры, либо когда цифр настолько много, что их невозможно преобразовать в целое число, не вызвав переполнения. В первом случае классический (но неэффективный и небезопасный) способ уведомить об ошибке заключается в том, чтобы присвоить переменной errno значение EINVAL (и вернуть некоторое бессмысленное значение, такое как 0). Во втором случае классический способ заключается в том, чтобы присвоить переменной errno значение ERANGE (и вернуть некоторое бессмысленное значение). Примерно такой подход используется в strtol.

Суть в том, что from_chars может столкнуться только с двумя, точно определенными ошибками. И их легко описать единым набором кодов, определяемым в <errno.h>. То есть чтобы перенести strtol из 1980-х в двадцать первый век, достаточно просто вернуть код ошибки непосредственно, а не косвенно, через errno. Именно это и делает стандартная библиотека. Классические значения из <errno.h> по-прежнему доступны как макросы, объявленные в <cerrno>, но начиная с C++11 они также определяются как строго типизированное перечисление в <system_error>, как демонстрирует следующий фрагмент:

```
namespace std {
  enum class errc {
    // "0" означает "нет ошибки"
```

```
      operation_not_permitted = EPERM,
      no_such_file_or_directory = ENOENT,
      no_such_process = ESRCH,
      // ...
      value_too_large = EOVERFLOW
    };
  } // namespace std
```

`std::from_chars` уведомляет об ошибках, возвращая структуру (`struct from_chars_result`) с переменной-членом типа `enum std::errc`, которая хранит 0 в случае отсутствия ошибок, или одно из двух возможных значений, определяющих вид ошибки.

А теперь вернемся к `fs::file_size`. Количество ошибок, с которыми может столкнуться `file_size`, намного, намного больше – только подумайте о разнообразии операционных систем и файловых систем, которые ими поддерживаются, учтите также, что некоторые файловые системы (такие как NFS) распределены в сетях разных типов, и вы поймете, что множество возможных ошибок невообразимо велико. Их можно было бы объявить в виде 78 стандартных вариантов в `sys::errc` (по одному для каждого POSIX-значения `errno`, кроме `EDQUOT`, `EMULTIHOP` и `ESTALE`), но тогда потерялся бы огромный объем информации. По крайней мере, один из выпавших вариантов POSIX (`ESTALE`) является допустимым кодом ошибки в `fs::file_size`! И конечно, сама файловая система могла бы возвращать свои коды ошибок; например, несмотря на наличие в POSIX стандартного кода для ошибки «слишком длинное имя файла», в POSIX нет кода ошибки «недопустимый символ в имени» (по причинам, с которыми мы познакомимся в следующем разделе). Возможно, файловая система пожелает сообщить именно эту ошибку, не заботясь о том, что `fs::file_size` может исчерпать доступный диапазон вариантов некоторого фиксированного типа перечисления.

Существенная проблема здесь в том, что не все ошибки, возвращаемые из `fs::file_size`, имеют один и тот же *источник*, а значит, их нельзя представить одним фиксированным типом (например, `std::errc`). Механизм исключений в C++ элегантно решает эту проблему; для разных уровней программы нормально и естественно возбуждать разные типы исключений. Если самый нижний уровень программы возбудит исключение `myfs::DisallowedCharacterInName`, самый верхний сможет перехватить его – по имени, по классу или по `...`. Если следовать правилу – любое исключение должно быть потомком `std::exception`, – тогда любой блок `catch` сможет вызвать `e.what()`, чтобы вывести более или менее осмысленное описание проблемы, независимо от ее природы.

Стандартная библиотека воплощает идею разнородных ошибок в базовом классе `std::error_category`:

```
namespace std {

class error_category {
```

```
public:
  virtual const char *name() const noexcept = 0;
  virtual std::string message(int err) const = 0;

  // другие виртуальные методы не показаны

  bool operator==(const std::error_category& rhs) const {
    return this == &rhs;
  }
};

} // namespace std
```

Класс `error_category` действует почти так же, как `memory_resource` из главы 8 «Диспетчеры памяти»; он определяет классический полиморфный интерфейс, а библиотеки, желающие использовать его, могут определить свои подклассы it. Обсуждая `memory_resource`, мы видели, что некоторые его подклассы являются глобальными синглтонами, а некоторые – нет. В случае с `error_category` *каждый* подкласс *должен* быть глобальным синглтоном, иначе он не будет работать.

Чтобы извлечь практическую пользу из ресурсов памяти, библиотека дает нам *контейнеры* (см. главу 4 «Зоопарк контейнеров»). На самом простом уровне контейнер – это указатель, представляющий некоторую область памяти, плюс *дескриптор ресурса памяти*, который знает, как освободить эту память. (Напомню, что этот дескриптор называется *диспетчером памяти*.)

Чтобы извлечь практическую пользу из подклассов `error_category`, библиотека дает нам `std::error_code`. На самом простом (и единственном в данном случае) уровне `error_code` – это значение типа `int`, представляющее код ошибки, плюс дескриптор `error_category`, который знает, как интерпретировать этот код:

```
namespace std {

class error_code {
  const std::error_category *m_cat;
  int m_err;
public:
  const auto& category() const { return m_cat; }
  int value() const { return m_err; }
  std::string message() const { return m_cat->message(m_err); }
  explicit operator bool() const { return m_err != 0; }

  // другие вспомогательные методы не показаны
};

} // namespace std
```

Вот как можно создать библиотечную подсистему для поддержки фиктивной файловой системы:

```
namespace FinickyFS {

enum class Error : int {
  success = 0,
  forbidden_character = 1,
  forbidden_word = 2,
  too_many_characters = 3,
};

struct ErrorCategory : std::error_category
{
  const char *name() const noexcept override {
    return "finicky filesystem";
  }

  std::string message(int err) const override {
    switch (err) {
      case 0: return "Success";
      case 1: return "Invalid filename";
      case 2: return "Bad word in filename";
      case 3: return "Filename too long";
    }
    throw Unreachable();
  }

  static ErrorCategory& instance() {
    static ErrorCategory instance;
    return instance;
  }
};

std::error_code make_error_code(Error err) noexcept {
  return std::error_code(int(err), ErrorCategory::instance());
}

} // namespace FinickyFS
```

Предыдущий пример определяет новую область ошибок, `FinickyFS::Error`, воплощенную как `FinickyFS::ErrorCategory::instance()`. Он позволяет создавать объекты типа `std::error_code`, как `make_error_code(FinickyFS::Error::forbidden_word)`.

Обратите внимание, что механизм аргумент-зависимого поиска (Argument-Dependent Lookup, ADL) правильно отыщет перегруженную версию `make_error_code` без нашей помощи. `make_error_code` является точкой настройки, точно так же как `swap`: просто определите функцию с этим именем в пространстве имен с вашим перечислением, а все остальное компилятор сделает самостоятельно.

```
// Ошибка благополучно укладывается в статически типизированный
// объект с семантикой значения std::error_code ...
std::error_code ec =
  make_error_code(FinickyFS::Error::forbidden_word);

// ...и его "строка с описанием" остается
// доступной, так же как в случае динамически
// типизированного исключения!
assert(ec.message() == "Bad word in filename");
```

Теперь у нас есть возможность передавать коды `FinickyFS::Error` через систему без потери информации – заворачивая их в тривиально копируемые объекты `std::error_code` – и извлекать их обратно на верхнем уровне. Написав эти слова, я задумался: звучит как волшебство, как обработка исключений без исключений! Но, как вы только что видели, реализовать все это не составляет труда.

Коды ошибок и условия ошибок

Обратите внимание, что `FinickyFS::Error` явно преобразуется в `std::error_code`; в последнем примере мы использовали синтаксис `make_error_code(FinickyFS::Error::forbidden_word)`, чтобы сконструировать начальный объект `error_code`. Мы можем сделать `FinickyFS::Error` более удобным для программиста, если сообщим `<system_error>`, что `FinickyFS::Error` можно неявно преобразовать в `std::error_code`, как показано ниже:

```
namespace std {
  template<>
  struct is_error_code_enum<::FinickyFS::Error> : true_type {};
} // namespace std
```

Будьте внимательны, повторно открывая пространство имен `std`, – помните, что это должно делаться за пределами любого другого пространства имен! Иначе будет создано вложенное пространство имен, такое как `FinickyFS::std`. В данном конкретном случае, если вы ошибетесь, компилятор известит вас при попытке специализировать несуществующим типом `FinickyFS::std::is_error_code_enum`. Если вы повторно открываете пространство имен `std`, только чтобы специализировать тот или иной шаблон (и не нарушаете синтаксис специализации шаблонов), можете не беспокоиться, что ошибка останется *незамеченной*.

После специализации `std::is_error_code_enum` вашим типом перечисления обо всем остальном библиотека позаботится сама:

```
class error_code {
  // ...
  template<
    class E,
    class = enable_if_t<is_error_code_enum_v<E>>
```

```
    >
    error_code(E err) noexcept {
      *this = make_error_code(err);
    }
};
```

Неявное преобразование, показанное в предыдущем примере, обеспечивает удобный синтаксис, например прямое сравнение посредством ==, но, так как каждый объект `std::error_code` несет информацию об области, которой принадлежит ошибка, сравнение выполняется с проверкой типов. Равенство значений объектов `error_code` зависит не только от их *целочисленных значений*, но также от *адресов*, связанных с ними синглтонов `error-category`.

```
std::error_code ec = FinickyFS::Error::forbidden_character;

  // Сравнения выполняются с учетом типов.
assert(ec == FinickyFS::Error::forbidden_character);
assert(ec != std::io_errc::stream);
```

Специализация `is_error_code_enum<X>` может пригодиться, если вы собираетесь часто присваивать значения типа `X` переменным типа `std::error_code` или возвращать их из функций с типом возвращаемого значения `std::error_code`. Иными словами, специализация полезна, если ваш тип `X` действительно представляет источник ошибок – так сказать, сторону уравнения, возбуждающую исключения. А как насчет другой стороны, перехватывающей эти исключения? Допустим, вы заметили, что написали следующую функцию и еще несколько похожих на нее:

```
bool is_malformed_name(std::error_code ec) {
  return (
    ec == FinickyFS::Error::forbidden_character ||
    ec == FinickyFS::Error::forbidden_word ||
    ec == std::errc::illegal_byte_sequence);
}
```

Эта функция определяет *унарный предикат* для всей вселенной кодов ошибок; она возвращает `true` для любого кода ошибки, с которым связано понятие неправильно сформированного имени, с точки зрения нашей библиотеки FinickyFS. Мы можем просто поместить эту функцию прямо в нашу библиотеку, как `FinickyFS::is_malformed_name()` – и фактически я советую поступать именно так, – но стандартная библиотека реализует другой возможный подход. Вместо `error_code` можно определить `error_condition`, например:

```
namespace FinickyFS {

enum class Condition : int {
  success = 0,
  malformed_name = 1,
```

```
};

struct ConditionCategory : std::error_category {
  const char *name() const noexcept override {
    return "finicky filesystem";
  }

  std::string message(int cond) const override {
    switch (cond) {
      case 0: return "Success";
      case 1: return "Malformed name";
    }
    throw Unreachable();
  }

  bool equivalent(const std::error_code& ec, int cond) const
  noexcept override {
    switch (cond) {
      case 0: return !ec;
      case 1: return is_malformed_name(ec);
    }
    throw Unreachable();
  }

  static ConditionCategory& instance() {
    static ConditionCategory instance;
    return instance;
  }
};

std::error_condition make_error_condition(Condition cond) noexcept
{
  return std::error_condition(int(cond),
  ConditionCategory::instance());
}

} // namespace FinickyFS

namespace std {
  template<>
  struct is_error_condition_enum<::FinickyFS::Condition> : true_type
  {};
} // namespace std
```

После этого вызов `FinickyFS::is_malformed_name(ec)` можно заменить сравнением (`ec == FinickyFS::Condition::malformed_name`), как показано ниже:

```
std::error_code ec = FinickyFS::Error::forbidden_word;

  // Правая сторона неявно преобразуется в error_code
```

```
assert(ec == FinickyFS::Error::forbidden_word);

  // Правая сторона неявно преобразуется в error_condition
assert(ec == FinickyFS::Condition::malformed_name);
```

Однако так как мы не реализовали функцию `make_error_code(FinickyFS::Condition)`, будет непросто сконструировать объект `std::error_code`, хранящий одно из этих условий. Но это нормально; перечисления условий предназначены для проверки на стороне, перехватывающей исключения, а не для преобразования в `error_code` на стороне, возбуждающей их.

Стандартная библиотека предоставляет два типа перечислений с кодами ошибок (`std::future_errc` и `std::io_errc`) и один тип перечисления с условиями (`std::errc`). Именно так – `std::errc` в действительности перечисляет *условия*, а не *коды* ошибок! То есть было бы неправильно пытаться присвоить код ошибки POSIX объекту `std::error_code`; это условия, предназначенные для проверки на стороне, перехватывающей исключения, а не возбуждающей их. К сожалению, стандартная библиотека проявляет непоследовательность в этом отношении, по меньшей мере дважды. Во-первых, как мы видели, `std::from_chars` возвращает значение типа `std::errc` (что вдвойне неудобно; правильнее было бы возвращать `std::error_code`). Во-вторых, существует функция `std::make_error_code(std::errc)`, вносящая путаницу в семантическое пространство, хотя в действительности должна существовать (и существует) только `std::make_error_condition(std::errc)`.

Возбуждение ошибок с std::system_error

До сих пор мы рассматривали `std::error_code` – отличную альтернативу использованию механизма обработки исключений в C++. Но иногда приходится смешивать библиотеки, возбуждающие и не возбуждающие исключения, на разных уровнях системы. Стандартная библиотека поможет вам решить, по крайней мере, половину этой задачи. `std::system_error` – конкретный тип исключения, производный от `std::runtime_error`, который имеет достаточно места для хранения одного `error_code`. То есть если вы пишете библиотеку, использующую исключения вместо `error_code`, которая может получить `error_code`, указывающий на ошибку на более низком уровне системы, удачным решением будет завернуть этот `error_code` в объект `system_error` и возбудить исключение.

```
// Нижний уровень возвращает коды ошибок в error_code.
uintmax_t file_size(const fs::path& p,
    std::error_code& ec) noexcept;

// Мой уровень, возбуждающий исключения.
uintmax_t file_size(const fs::path& p)
{
  std::error_code ec;
  uintmax_t size = file_size(p, ec);
```

```
  if (ec) {
    throw std::system_error(ec);
  }
  return size;
}
```

Противоположная ситуация – когда вы пишете библиотеку, не возбуждающую исключений, но исключения могут порождаться на более низком уровне. В этом случае стандартная библиотека почти ничего не предлагает вам в помощь. Но вы легко можете сами написать код, разворачивающий error_code:

```
// Нижний уровень возбуждает исключения.
uintmax_t file_size(const fs::path& p);

// Мой уровень, использующий error_code.
uintmax_t file_size(const fs::path& p,
    std::error_code& ec) noexcept
{
  uintmax_t size = -1;
  try {
    size = file_size(p);
  } catch (...) {
    ec = current_exception_to_error_code();
  }
  return size;
}
```

Предыдущий фрагмент вызывает нестандартную функцию current_exception_to_error_code(), которую можно написать самим, например:

```
namespace detail {

enum Error : int {
  success = 0,
  bad_alloc_thrown = 1,
  unknown_exception_thrown = 2,
};

struct ErrorCategory : std::error_category {
  const char *name() const noexcept override;
  std::string message(int err) const override;
  static ErrorCategory& instance();
};

std::error_code make_error_code(Error err) noexcept {
  return std::error_code(int(err), ErrorCategory::instance());
}

} // namespace detail

std::error_code current_exception_to_error_code()
```

```
{
  try {
    throw;
  } catch (const std::system_error& e) {
    // также перехватывает std::ios_base::failure
    // и fs::filesystem_error
    return e.code();
  } catch (const std::future_error& e) {
    // необычные исключения
    return e.code();
  } catch (const std::bad_alloc&) {
    // bad_alloc часто представляет особый интерес
    return detail::bad_alloc_thrown;
  } catch (...) {
    return detail::unknown_exception_thrown;
  }
}
```

На этом мы завершаем экскурс в мир <system_error> и продолжим обсуждение <filesystem>.

Файловые системы и пути

В главе 9 «Потоки ввода/вывода» мы обсудили идею файловых дескрипторов POSIX. Файловый дескриптор представляет источник или приемник данных, который можно настроить на чтение и/или запись. Часто, но не всегда, он соответствует файлу на диске. (Напомню, что файловый дескриптор с номером 1 ссылается на стандартный вывод stdout, который обычно подключен к экрану монитора. Файловые дескрипторы могут также ссылаться на сетевые сокеты, устройства, такие как /dev/random, и т. д.)

Кроме того, файловые дескрипторы POSIX, <stdio.h> и <iostream> имеют непосредственное отношение к *содержимому* файла на диске – последовательности байтов, составляющей *содержимое* файла. Файл в представлении *файловой системы* имеет множество более существенных атрибутов, которые недоступны через API чтения/записи в файлы. Используя инструменты, описанные в главе 9 «Потоки ввода/вывода», нельзя определить принадлежность файла или дату/время последнего изменения; с их помощью также нельзя определить количество файлов в заданном каталоге. Цель <filesystem> – дать нашим программам на C++ возможность обращаться с этими атрибутами файловой системы переносимым способом.

Начнем с самого начала. Что такое файловая система? Файловая система – это абстракция сопоставления путей с файлами посредством элементов каталога. Лучше понять суть сказанного вам поможет диаграмма на рис. 12.1.

Вверху на диаграмме (рис. 12.1) изображен абстрактный мир «имен». У нас имеется отображение этих имен (таких как speech.txt) в конкретные структуры, которые в терминологии POSIX называются *индексными узлами* (inode).

Термин «индексный узел» не используется стандартом C++ – он использует обобщенный термин «файл», – но я буду использовать термин «индексный узел» для большей точности. Каждый индексный узел содержит полный набор атрибутов, описывающих единственный файл на диске: его владельца, дату последнего изменения, тип и т. д. Но самое главное индексный узел определяет размер файла и содержит указатель на фактическое содержимое (подобно тому, как `std::vector` или `std::list` хранит указатель на свое содержимое). Точное представление индексных узлов и блоков данных на диске зависит от типа файловой системы. К наиболее распространенным файловым системам можно отнести ext4 (в Linux), HFS+ (в OS X) и NTFS (в Windows).

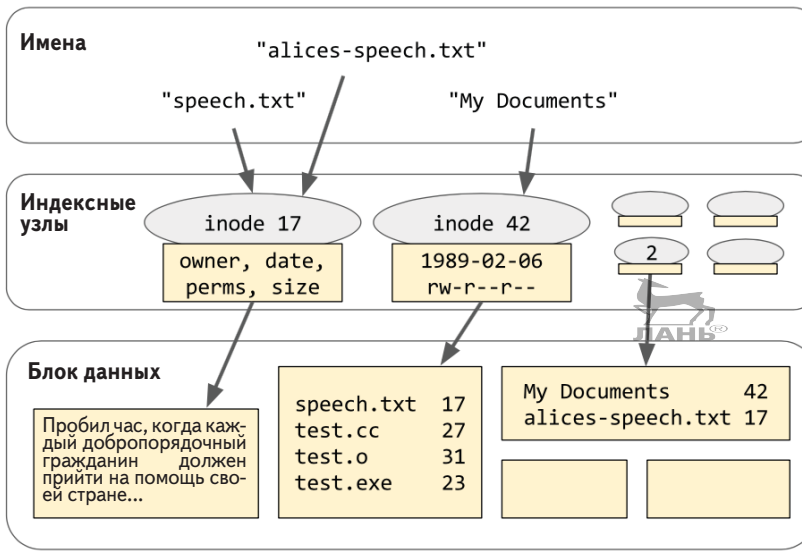


Рис. 12.1. Файловая система

Обратите внимание, что некоторые блоки на этой диаграмме содержат данные, которые являются табличным представлением имен и соответствующих им номеров индексных узлов. Круг замкнулся! Итак, *каталог* – это индексный узел определенного типа, содержащий табличное представление с именами и номерами индексных узлов. В каждой файловой системе имеется один специальный индексный узел, который называется *корневым каталогом*.

Допустим, что корневой узел с меткой «2» на нашей диаграмме является корневым каталогом. Тогда мы можем однозначно идентифицировать файл с текстом «Пробил час...» цепочкой имен, ведущей к нему от корневого каталога. Например, `/My Documents/speech.txt`: она начинается с корневого каталога, имя `My Documents` отображается в индексный узел 42, который также является каталогом и отображает имя `speech.txt` в индексный узел 17, представляющий обычный файл, хранящий на диске текст «Пробил час...». Для разделения

имен в этой цепочке мы используем символы слеша (/) и добавляем один слеш в начало цепочки – он соответствует корневому каталогу. (В Windows каждый раздел, или диск, имеет свой корневой каталог. Поэтому вместо `/My Documents/speech.txt` путь к файлу записывается немного иначе – `c:/My Documents/speech.txt`, – чтобы показать, что путь начинается в корневом каталоге на диске C.)

Напротив, `"/alices-speech.txt"` – это путь, ведущий к файлу прямо из корневого каталога к индексному узлу 17. Мы говорим, что эти два пути (`"/My Documents/speech.txt"` и `"/alicesspeech. txt"`) являются *жесткими ссылками* на один и тот же индексный узел, а значит, соответствуют одному и тому же файлу. Некоторые файловые системы (такие как FAT,используемая на многих USB-накопителях) не поддерживают возможности создания нескольких жестких ссылок на один и тот же файл. Файловая система, поддерживающая множественные жесткие ссылки, должна подсчитывать количество ссылок на каждый индексный узел, чтобы знать, когда можно безопасно удалить и освободить индексный узел, подобно тому, как ведется подсчет ссылок в `shared_ptr` (см. главу 6 «Умные указатели»).

Когда мы вызываем библиотечную функцию, такую как `open` или `fopen`, чтобы «открыть файл», она выполняет следующую процедуру, скрытую глубоко в недрах файловой системы: извлекает имя файла, переданное ей, и преобразует его в путь – разбивает строку по символам слеша и спускается по дереву каталогов в файловой системе, пока не достигнет индексного узла с требуемым файлом (или не обнаружит, что такого файла нет). Обратите внимание, что по достижении индексного узла отпадает необходимость в имени файла, потому что он может иметь столько же имен, сколько имеется жестких ссылок.

Представление путей в C++

Все функции, представленные в главе 9 «Потоки ввода/вывода», с параметром «имя файла» (то есть путь) благополучно принимают этот путь в виде `const char *`. Но в библиотеке `<filesystem>` картина усложняется, в основном из-за Windows.

Все файловые системы, соответствующие стандарту POSIX, хранят имена (такие как `speech.txt`) в виде простых строк байтов. POSIX выдвигает единственное требование – имена не должны содержать символы `'\0'` и `'/'` (потому что последний используется как разделитель компонентов пути). Строка `"\xC1.h"` считается в POSIX вполне допустимым именем файла, даже притом, что она не является допустимой строкой UTF-8 и ASCII, а вид такого имени на экране, как его выведет команда `ls .`, полностью зависит от региональных настроек. В конце концов, это всего лишь строка из трех байтов, ни один из которых не соответствует символу `'/'`.

В Window, напротив, встроенные API доступа к файлам, такие как `CreateFileW`, всегда хранят имена в кодировке UTF-16. То есть пути в Windows по определению всегда являются допустимыми строками Юникода. Это важное

философское различие между POSIX и NTFS! Повторю еще раз и помедленнее: имена файлов в POSIX – это *строки байтов*. Имена файлов в Windows – это *строки символов Юникода*.

Если следовать основному принципу из главы 9 «Потоки ввода/вывода», что все в мире должно быть представлено в кодировке UTF-8, тогда различия между POSIX и Windows окажутся вполне преодолимыми, возможно, даже незначительными. Но если вам доведется решать проблемы с необычными именами файлов в той или в другой системе, имейте в виду: имена файлов в POSIX – это строки байтов, а в Windows – это строки символов.

Поскольку Windows API оперирует строками UTF-16 (`std::u16string`), а POSIX API – строками байтов (`std::string`), ни одно из представлений не может в точности соответствовать требованиям кросс-платформенности. Поэтому в `<filesystem>` добавлен новый тип: `fs::path`. (Напомню, что мы используем псевдоним имени пространства имен, то есть в действительности `std::filesystem::path`.) Определение типа `fs::path` выглядит примерно так:

```
class path {
public:
  using value_type = std::conditional_t<
    IsWindows, wchar_t, char
  >;
  using string_type = std::basic_string<value_type>;

  const auto& native() const { return m_path; }
  operator string_type() const { return m_path; }
  auto c_str() const { return m_path.c_str(); }

  // другие конструкторы и методы опущены
private:
  string_type m_path;
};
```

Обратите внимание, что `fs::path::value_type` – это тип `wchar_t` в Windows, даже притом, что на эту роль больше подходит тип `char16_t` из C++11. Это всего лишь наследие исторических корней библиотеки Boost. В этой главе, когда мы будем говорить о `wchar_t`, вы можете смело предполагать, что речь идет об UTF-16, и наоборот.

Переносимый код должен преобразовывать возвращаемые значения `fs::path` всех функций в строки. Например, обратите внимание, что функция `path.c_str()` возвращает не `const char *`, а `const value_type *`!

```
fs::path p("/foo/bar");

const fs::path::value_type *a = p.c_str();
  // Переносимый для любой системы.

const char *b = p.c_str();
```

```
    // Допустим в POSIX; ошибка компиляции в Windows.

std::string s = p.u8string();
const char *c = s.c_str();
  // Допустим в POSIX и в Windows.
  // В Windows выполняется преобразование 16-to-8.
```


Вариант с `c` в предыдущем примере гарантированно компилируется, но его поведение отличается на разных платформах: на платформах POSIX вы получите простую строку байтов, а в Windows выполнится дорогостоящее преобразование `path.native()` из UTF-16 в UTF-8 (именно то, что запрошено, но программа может работать быстрее, если найти способ избежать запроса).

`fs::path` имеет шаблонный конструктор, который может создать экземпляр `path` практически из любого аргумента. Аргумент может быть последовательностью символов любого типа (`char`, `wchar_t`, `char16_t` или `char32_t`), и эта последовательность может быть представлена указателем на строку, завершающуюся нулевым символом, итератором на такую же строку, `basic_string`, `basic_string_view` или парой итераторов. Как обычно, я упоминаю об этом разнообразии перегруженных версий не потому, что вы должны использовать их в своей практике, но затем, чтобы вы знали, как их избегать.

Стандарт также предоставляет свободную функцию `fs::u8path("path")`, которая в действительности является всего лишь синонимом для `fs::path("path")`, но может служить напоминанием, что передаваемая строка должна быть представлена в кодировке UTF-8. Лично я советую не использовать `u8path`.

Все это только кажется пугающим. Имейте в виду, что если использовать имена файлов в кодировке ASCII, вам не придется беспокоиться о проблемах кодирования; а если избегать методов `path.native()` и `path.c_str()` и неявного преобразования в `fs::path::string_type`, вам почти не придется беспокоиться о переносимости.

Операции с путями

Получив объект пути, вы получаете возможность обращаться ко множеству его компонентов с использованием разделителя-слеша. В следующем фрагменте каждый идентификатор `x` (кроме самого `path`) представляет значение, возвращаемое функцией-членом `path.x()`:

```
assert(root_path == root_name / root_directory);
assert(path == root_name / root_directory / relative_path);
assert(path == root_path / relative_path);

assert(path == parent_path / filename);
assert(filename == stem + extension);

assert(is_absolute == !is_relative);
if (IsWindows) {
```

```
    assert(is_relative == (root_name.empty() ||
                           root_directory.empty()));
  } else {
    assert(is_relative == (root_name.empty() &&
           root_directory.empty()));
  }
```

Например, для пути `p = "c:/foo/hello.txt"` мы получим `p.root_name() == "c:"`, `p.root_directory() == "/"`, `p.relative_path() == "foo/hello.txt"`, `p.stem() == "hello"` и `p.extension() == ".txt"`. По крайней мере, такие результаты мы получили в Windows! Обратите внимание, что в Windows абсолютный путь должен содержать не только корневой каталог, но также имя диска (ни `"c:foo/hello.txt"`, ни `"/foo/hello.txt"` не являются абсолютными путями), тогда как в POSIX, где имя диска отсутствует, абсолютный путь должен начинаться только с символа слеша (`"/foo/hello.txt"` – абсолютный путь, а `"c:foo/hello.txt"` – относительный, начинающийся в каталоге с необычным именем `"c:foo"`).

В последнем примере мы использовали `operator/` для объединения компонентов пути. Для этой цели `fs::path` поддерживает `operator/` и `operator/=`, и их поведение почти в точности соответствует нашим ожиданиям – они объединяют фрагменты пути, добавляя символ слеша между ними. Если символ слеша вам не нравится, объединяйте компоненты с помощью `operator+=`. К сожалению, в стандартной библиотеке C++17 отсутствует `operator+` для путей, но его легко добавить в виде свободной функции, как показано ниже:

```
static fs::path operator+(fs::path a, const fs::path& b)
{
  a += b;
  return a;
}
```

Пути также поддерживают конкатенацию с и без слешей в виде функций-членов с именами `path.concat("foo")` (без слешей) и `path.append("foo")` (со слешами), способными вызвать путаницу. Будьте внимательны, не перепутайте! Лично я советую никогда не использовать эти функции-члены; всегда применяйте операторы (включая собственную реализацию `operator+`, представленную выше).

Последний источник возможной путаницы в `fs::path` – методы `begin` и `end` действуют точно так же, как одноименные методы типа `std::string`. Но, в отличие от `std::string`, итерации выполняются не по символам, а по именам! Взгляните на следующий пример:

```
fs::path p = "/foo/bar/baz.txt";
std::vector<fs::path> v(p.begin(), p.end());
assert((v == std::vector<fs::path>{
  "/", "foo", "bar", "baz.txt"
}));
```

У вас едва ли будут веские причины организовать обход элементов абсолютного пути `fs::path`. Однако итерации по `p.relative_path().parent_path()` – где каждый элемент гарантированно будет именем каталога – могут пригодиться в некоторых необычных ситуациях.

Получение информации о файлах с directory_entry

Внимание! `directory_entry` – самая малопонятная часть библиотеки `<filesystem>` в C++17. Все, о чем рассказывается далее, не реализовано ни в Boost, ни в `<experimental/filesystem>`.

Извлечение метаданных о файле из его индексного узла выполняется посредством объекта типа `fs::directory_entry`. Если вы знакомы с инструментами POSIX для получения метаданных, вообразите, что `fs::directory_entry` содержит члены с типами `fs::path` и `std::optional<struct stat>`. Вызов `entry.refresh()` фактически действует точно так же, как POSIX-функция `stat()`; а методы доступа, такие как `entry.file_size()`, неявно вызывают `stat()`, только если член `optional` пока не получил значения. Простое создание экземпляра `fs::directory_entry` не посылает запрос в файловую систему; библиотека ждет, когда вы зададите конкретный вопрос. Конкретный вопрос, такой как `entry.file_size()`, может заставить библиотеку послать запрос в файловую систему или (если член `optional` уже получил значение) использовать кешированное значение, оставшееся от предыдущего запроса.

```
fs::path p = "/tmp/foo/bar.txt";
fs::directory_entry entry(p);
  // Здесь еще нет обращения к файловой системе.

while (!entry.exists()) {
  std::cout << entry.path() << " does not exist yet\n";
  std::this_thread::sleep_for(100ms);
  entry.refresh();
    // Без refresh() цикл будет выполняться вечно.
}
  // Если файл удалить в этот момент, следующая строка
  // может вывести устаревшие данные из кеша или может
  // попытаться обновить кеш и возбудит исключение.
std::cout << entry.path() << " has size "
          << entry.file_size() << "\n";
```

Прежде для достижения этой же цели вызывалась функция `fs::status("path")` или `fs::symlink_status("path")`, которая возвращает экзем-

пляр класса `fs::file_status`, из которого потом можно получить информацию, применяя громоздкие операции, такие как `status.type() == fs::file_type::directory`. Я не советую пользоваться `fs::file_status`, лучше используйте `entry.is_directory()`. Впрочем, мазохисты могут по-прежнему извлекать экземпляр `fs::file_status` непосредственно из `directory_entry`: `entry.status()` – это эквивалент для `fs::status(entry.path())`, а `entry.symlink_status()` – эквивалент для `fs::symlink_status(entry.path())`, который также является чуть более быстрым эквивалентом выражения `fs::status(entry.is_symlink() ? fs::read_symlink(entry.path()) : entry.path())`.

Кстати, свободная функция `fs::equivalent(p, q)` поможет вам узнать, являются ли два пути жесткими ссылками на один и тот же индексный узел; а `entry.hard_link_count()` вернет общее количество жестких ссылок на данный конкретный индексный узел. (Единственный способ узнать имена этих жестких ссылок – выполнить обход всей файловой системы; но имейте в виду, что ваша учетная запись может иметь недостаточно привилегий для этого.)

Обход каталогов с directory_iterator

Итератор `fs::directory_iterator` делает именно то, что предполагает его имя. Объект этого типа позволяет выполнить обход содержимого единственного каталога, поэлементно:

```
fs::path p = fs::current_path();
  // Обход текущего каталога.
for (fs::directory_entry entry : fs::directory_iterator(p)) {
  std::cout << entry.path().string() << ": "
            << entry.file_size() << " bytes\n";
}
```

Кстати, обратите внимание на вызов функции `entry.path().string()` в предыдущем примере. Это необходимо, потому что `operator<<` проявляет странное поведение в отношении объектов `path` – он всегда выводит их, как если бы вызывалась `std::quoted(path.string())`. Если вы хотите получить путь без дополнительных кавычек, всегда преобразуйте его в `std::string` перед выводом. (Аналогично `std::cin >> path` не позволит вам получить путь от пользователя, но это меньшая неприятность, так как вы никогда не должны использовать `operator>>` в таких случаях. Больше информации о лексемизации и парсинге ввода пользователя вы найдете в главе 9 «Потоки ввода/вывода» и в главе 10 «Регулярные выражения».)

Рекурсивный обход каталогов

Чтобы выполнить рекурсивный обход всего дерева каталогов в стиле `os.walk()` в языке Python, можно воспользоваться следующей рекурсивной функцией, основанной на предыдущем примере:

```
template<class F>
void walk_down(const fs::path& p, const F& callback)
{
  for (auto entry : fs::directory_iterator(p)) {
    if (entry.is_directory()) {
      walk_down(entry.path(), callback);
    } else {
      callback(entry);
    }
  }
}
```

Или просто использовать `fs::recursive_directory_iterator`:

```
template<class F>
void walk_down(const fs::path& p, const F& callback)
{
  for (auto entry : fs::recursive_directory_iterator(p)) {
    callback(entry);
  }
}
```

Конструктор `fs::recursive_directory_iterator` имеет необязательный аргумент типа `fs::directory_options`, который влияет на точный характер рекурсии. Например, можно передать значение `fs::directory_options::follow_directory_symlink`, чтобы обеспечить следование по символическим ссылкам, но имейте в виду, что это отличный способ попасть в бесконечный цикл, если злонамеренный пользователь создаст символическую ссылку обратно на родительский каталог.

Изменение файловой системы

Большинство инструментов в заголовке `<filesystem>` предназначено для получения информации о файловой системе, а не для ее изменения. Но среди черепков скрыто несколько драгоценных камней. Многие из следующих функций, похоже, проектировались как реализация классических утилит командной строки POSIX на переносимом C++:

- `fs::copy_file(old_path, new_path)`: копирует файл `old_path` в новый файл (то есть в новый индексный узел) `new_path`, подобно команде `cp -n`; завершается с ошибкой, если `new_path` уже существует;
- `fs::copy_file(old_path, new_path, fs::copy_options::overwrite_existing)`: копирует `old_path` в `new_path`, затирает `new_path`, если возможно; завершается с ошибкой, если `new_path` существует и не является обычным файлом или если `new_path` и `old_path` ссылаются на один и тот же файл;

- `fs::copy_file(old_path, new_path, fs::copy_options::update_existing)`: копирует `old_path` в `new_path`, затирает `new_path`, только если он старше, чем `old_path`;
- `fs::copy(old_path, new_path, fs::copy_options::recursive | fs::copy_options::copy_symlinks)`: копирует все содержимое каталога `old_path` в `new_path`, подобно команде `cp -R`;
- `fs::create_directory(new_path)`: создает каталог, подобно команде `mkdir`;
- `fs::create_directories(new_path)`: создает каталог, подобно команде `mkdir -p`;
- `fs::create_directory(new_path, old_path)` (обратите внимание на обратный порядок аргументов!): создает каталог, но копирует атрибуты старого каталога `old_path`;
- `fs::create_symlink(old_path, new_path)`: создает символическую ссылку из `new_path` на `old_path`;
- `fs::remove(path)`: удаляет файл или пустой каталог, подобно команде `rm`;
- `fs::remove_all(path)`: удаляет файл или пустой каталог, подобно команде `rm -r`;
- `fs::rename(old_path, new_path)`: переименовывает файл или каталог, подобно команде `mv`;
- `fs::resize_file(path, new_size)`: увеличивает (дополняя нулями) или усекает обычный файл.

Получение информации о диске

В продолжение разговора об утилитах командной строки хотелось бы также отметить возможность узнать, насколько заполнена файловая система. Эту функцию выполняет утилита `df -h`, а также POSIX-функция `statvfs`. В C++17 аналогичную роль играет `fs::space("path")`, которая возвращает (по значению) структуру типа `fs::space_info`:

```
struct space_info {
  uintmax_t capacity;
  uintmax_t free;
  uintmax_t available;
};
```

Каждое из этих полей содержит размер в байтах, и для них выполняется условие `available <= free <= capacity`. Разница между `available` (доступно) и `free` (свободно) состоит в том, что в некоторых файловых системах часть свободного пространства резервируется для нужд суперпользователя `root`, а в других доступное пространство может ограничиваться дисковыми квотами.

Итоги

Используйте псевдонимы пространств имен, чтобы сэкономить на нажатиях клавиш и получить возможность прозрачно переключать альтернативные реализации, такие как Boost.

`std::error_code` дает очень удобный способ передачи целочисленных кодов ошибок вверх по стеку без возбуждения исключений; вы можете использовать этот тип в окружениях, где исключения не приветствуются. (В таком случае это самый главный урок данной главы!) Библиотека `<filesystem>` поддерживает два API – с исключениями и без исключений, – но оба используют динамическую память (и `fs::path` как словарный тип). Единственная причина использовать API, не возбуждающий исключений, – нежелание «применять исключения для управления порядком выполнения».

`std::error_condition` – это лишь синтаксический сахар для «перехвата» кодов ошибок; старайтесь не использовать этот тип.

Путь `path` включает элементы `root_name`, `root_directory` и `relative_path`; последний состоит из *имен*, разделенных слешами. В POSIX *имя* – это строка простых байтов; в Windows – строка символов Юникода. Тип `fs::path` стремится использовать тип строки, наиболее подходящий для каждой платформы. Чтобы избежать проблем переносимости, остерегайтесь функции `path.c_str()` и неявного преобразования в `fs::path::string_type`.

Каталоги хранят отображения имен в индексные узлы inode (которые в стандарте C++ называются «файлами»). Обход содержимого каталога в C++ можно выполнить с помощью `fs::directory_iterator` и получить объекты `fs::directory_entry`; методы `fs::directory_entry` позволят вам узнать номер соответствующего индексного узла. Чтобы обновить информацию об индексном узле, достаточно вызвать `entry.refresh()`.

`<filesystem>` включает также целую коллекцию свободных функций для создания, копирования, переименования, удаления файлов и каталогов и изменения их размеров. И наконец, имеется функция, позволяющая определить общую емкость файловой системы.

Многое из того, что обсуждалось в этой главе (по меньшей мере, в части, относящейся к заголовку `<filesystem>`), появилось только в стандарте C++17, и по этой причине на данный момент многое еще не реализовано производителями компиляторов. Используйте эти новые возможности с осторожностью.

Предметный указатель

Книги издательства «ДМК Пресс» можно заказать в торгово-издательском холдинге «Планета Альянс» наложенным платежом, выслав открытку или письмо по почтовому адресу: **115487, г. Москва, 2-й Нагатинский пр-д, д. 6А**.

При оформлении заказа следует указать адрес (полностью), по которому должны быть высланы книги; фамилию, имя и отчество получателя. Желательно также указать свой телефон и электронный адрес.

Эти книги вы можете заказать и в интернет-магазине: **www.a-planeta.ru**.

Оптовые закупки: **тел. +7 (499) 782-38-89**.

Электронный адрес: **books@alians-kniga.ru**.

Артур О'Двайр

Осваиваем C++17 STL

Используйте компоненты стандартной библиотеки в C++17 в полной мере

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Перевод с английского *Киселев А. Н.*
Корректор *Синяева Г. И.*
Верстка *Паранская Н. В.*
Дизайн обложки *Докучаева А. Е.*

Формат 70×100 1/16.
Печать цифровая. Усл. печ. л. 28,6.
Тираж 200 экз.

Веб-сайт издательства: www.dmkpress.com